

University of Warwick institutional repository: <http://go.warwick.ac.uk/wrap>

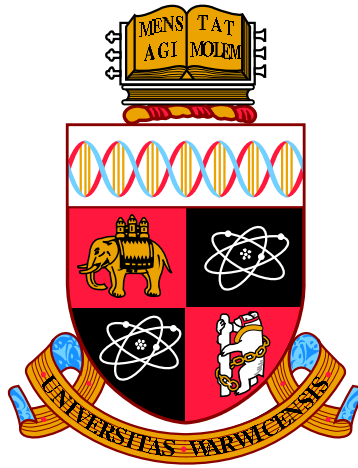
**A Thesis Submitted for the Degree of PhD at the University of Warwick**

<http://go.warwick.ac.uk/wrap/49186>

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



# Algorithms for Polycyclic-by-Finite Groups

by

**Shavak Sinanan**

**Thesis**

Submitted to The University of Warwick

for the degree of

**Doctor of Philosophy**

**Mathematics Institute**

July, 2011

THE UNIVERSITY OF  
**WARWICK**

*To my parents,  
Avory & Ousha*

# Table of Contents

List of Tables	vii
List of Algorithms	viii
Listings	ix
Acknowledgements	x
Declaration	xiii
Abstract	xiv
Notation	xv
<b>1 Introduction</b>	<b>1</b>
1.1 The Class of Polycyclic-by-Finite Groups . . . . .	1
1.2 Overview of Project . . . . .	2
1.2.1 Objectives . . . . .	2
1.2.2 Results . . . . .	3
1.2.3 Notes and Assumptions . . . . .	5
<b>2 Background Material</b>	<b>6</b>
2.1 Permutation Groups . . . . .	6
2.1.1 Definitions and Notation . . . . .	7

2.1.2	Computation of Orbits and Stabilisers . . . . .	8
2.1.3	Bases and Strong Generating Sets . . . . .	11
2.2	Polycyclic Groups . . . . .	17
2.2.1	Finite Soluble Groups . . . . .	19
2.2.2	Infinite Polycyclic Groups . . . . .	21
2.3	Representation Theory and Extensions . . . . .	22
2.3.1	The Terminology of Representation Theory . . . . .	23
2.3.2	Semidirect Products, Complements, Derivations and First Co- homology Groups . . . . .	24
2.3.3	Extensions of Modules and The Second Cohomology Group . . . . .	26
<b>3</b>	<b>Multiplication</b>	<b>29</b>
3.1	Representation of Elements . . . . .	30
3.1.1	The Normal Form . . . . .	30
3.1.2	The Multiplication Problem . . . . .	31
3.2	The Multiplication Algorithm . . . . .	32
3.2.1	Strategy . . . . .	32
3.2.2	Data . . . . .	33
3.2.3	The Algorithm . . . . .	37
3.3	Applications . . . . .	44
3.3.1	Transversal Images . . . . .	45
3.3.2	Element Inversion . . . . .	45
3.3.3	Orders of Elements . . . . .	46
3.3.4	Transfer to the Category of Polycyclic Groups . . . . .	46
<b>4</b>	<b>Subgroups</b>	<b>50</b>
4.1	The Subgroup Generation Problem . . . . .	51
4.2	Construction . . . . .	52
4.2.1	The Extended Schreier–Sims Algorithm . . . . .	52

---

4.2.2	Data . . . . .	64
4.2.3	Membership Testing . . . . .	66
4.3	Applications . . . . .	67
4.3.1	The Soluble Radical . . . . .	67
4.3.2	Sylow Subgroups . . . . .	68
<b>5</b>	<b>Conjugacy</b>	<b>70</b>
5.1	The Centre . . . . .	71
5.2	The Conjugacy Problem . . . . .	75
5.2.1	Centralisers . . . . .	75
5.2.2	Conjugacy Testing . . . . .	80
<b>6</b>	<b>Conclusion</b>	<b>83</b>
6.1	Examples and Run-times . . . . .	83
6.2	Implementation . . . . .	101
6.2.1	List of Functions . . . . .	101
6.2.2	Technical Considerations . . . . .	102
6.2.3	Native Support . . . . .	103
6.3	Future Work . . . . .	104
<b>A</b>	<b>Package Documentation</b>	<b>106</b>
A.1	Introduction . . . . .	106
A.2	Getting Started . . . . .	106
A.3	Creation of a Group . . . . .	107
A.3.1	Construction Functions . . . . .	107
A.4	Basic Group Properties . . . . .	107
A.4.1	Infrastructure . . . . .	107
A.4.2	Numerical Invariants . . . . .	108
A.4.3	Predicates . . . . .	108

---

A.5	Elements . . . . .	108
A.5.1	Definition of Elements . . . . .	109
A.5.2	Arithmetic Operations on Elements . . . . .	109
A.5.3	Properties of Elements . . . . .	110
A.5.4	Predicates for Elements . . . . .	110
A.5.5	Set Operations . . . . .	111
A.6	Subgroups . . . . .	111
A.6.1	Definition of Subgroups by Generators . . . . .	112
A.6.2	Membership and Coercion . . . . .	112
A.6.3	Standard Subgroup Constructions . . . . .	112
A.6.4	Sylow Subgroups . . . . .	113
A.6.5	Centralisers . . . . .	113
A.7	Normal Subgroups and Subgroup Series . . . . .	113
A.7.1	Normal Structure . . . . .	113
A.7.2	Characteristic Subgroups . . . . .	113
A.7.3	Subgroup Series . . . . .	114
A.8	Conjugacy . . . . .	114
A.9	Transfer Between Group Categories . . . . .	114
A.9.1	Transfer to GrpPC or GrpGPC . . . . .	114
<b>B</b>	<b>Program Listing</b>	<b>116</b>
	<b>Bibliography</b>	<b>238</b>

# List of Tables

6.1	Run-times for perfect groups . . . . .	95
6.2	Run-times for subgroups of $\text{AGL}(3, 5)$ . . . . .	96
6.3	Run-times for subgroups of $\text{AGL}(4, 3)$ . . . . .	97
6.4	Run-times for reducible matrix groups . . . . .	98
6.5	Run-times for module extensions . . . . .	99
6.6	Run-times for finitely presented groups that map onto $A_5$ . . . . .	100



# List of Algorithms

3.1	Element Multiplication . . . . .	42
4.1	Sifting . . . . .	55
4.2	Standard version of the extended Schreier–Sims algorithm . . . . .	59
4.3	Normal closure version of the extended Schreier–Sims algorithm . . . . .	63
4.4	Base-preserving version of the extended Schreier–Sims algorithm . . . . .	65
5.1	Finding the centre . . . . .	74
5.2	Finding centralisers . . . . .	79
5.3	Testing Element Conjugacy . . . . .	82

# Listings

B.1	grputils.m . . . . .	116
B.2	permsv.m . . . . .	117
B.3	recfrmtdef.m . . . . .	122
B.4	permdata.m . . . . .	124
B.5	pcbfconstruct.m . . . . .	130
B.6	pcbfarithmetic.m . . . . .	159
B.7	pcbfcattrans.m . . . . .	173
B.8	pcbfschreiersims.m . . . . .	180
B.9	pcbfsubgroup.m . . . . .	197
B.10	pcbfderived.m . . . . .	213
B.11	pcbf Sylow.m . . . . .	218
B.12	pcbfcentre.m . . . . .	221
B.13	pcbfcentraliser.m . . . . .	225
B.14	pcbfconjugacy.m . . . . .	229
B.15	pcbfmain.m . . . . .	233

## Acknowledgements

First and foremost I would like to express my deepest gratitude to my supervisor, Prof. Derek Holt, who has, at all times, been a source of inspiration and encouragement. Derek has been an excellent supervisor, and is an outstanding mathematician. His keen insights and sharp observations have always kept me interested and motivated, and I owe every ounce of mathematical knowledge and skill that I have acquired over the last four years to him.

A key ingredient to success is a productive, challenging yet comfortable environment, and I am fortunate to have found that and so much more in, where I now call my mathematical home, the Warwick Mathematics Institute. The support provided by the Institute is all that one could hope for as a graduate student, and I wish to sincerely thank the staff for the magnificent job that they have done, and continue to do, in managing us mathematicians. Special mention must be made of the current and former Postgraduate Coordinators, Carole Fisher and Carol Wright, for whose patience and understanding over the last four years I am eternally grateful. I would also like to acknowledge the University of Warwick for the financial support that I have received via the Warwick Postgraduate Scholarship and the Chancellor's International Scholarship during my time as a doctoral candidate.

I must also express my gratitude to Prof. John Cannon of the Computational Algebra Group at the University of Sydney, for providing me with the opportunity to visit Sydney to collaborate directly with the Magma developers, and for his keen interest in my work.

Special acknowledgement must be made of those who have contributed to my earlier academic development, in particular I would like to mention Prof. Michael Vaughan-Lee of Christ Church College, Oxford, and Prof. E. J. Farrell of the University of the West Indies, St. Augustine.

As I near the end of my time at Warwick, I think about the reasons why I am where I am now, and there are just two: Avory and Ousha. To my parents: your unwavering support at all stages of my life means more to me than you shall ever know, and you will *always* be the reason why I am where I am.

A famous quote of William Butler Yeats goes: “Think where man’s glory most begins and ends, and say my glory was I had such friends.” To those most important to me:

Rajiv ‘Fliz’ Sinanan — “You’re the best man I know.”

Rishi ‘Ding’ Ramsumair — C \_ c a \_ \_ l \_ ×3. SBW. Blauuw.

Travers ‘JT’ Sinanan — “Be resourceful man. Bring me some champagne and oysters.”

Kerwin ‘The Prophet’ Cumberbatch — Salaam brother Numsi.

Avinash ‘T.C.’ Ramdass — The real man in short pants.

Shivana ‘Tea Poog’ Maharaj — For F.R.I.E.N.D.S and crêpes.

Mandana ‘Mana Joon’ Namdar — For halloumi cheese distractions!

Tasneem ‘Joggers’ Sharafally — Gorillas, Shirts and Permitted.

Andrew ‘Koopas Troopa’ Ferguson — Red shell first, cigar later.

Shivani ‘Nanny Goat’ Jacelon — You’re ugly.

Sasha ‘The Popo’ Nath — For Jay Sean and Apple Sourz bonding.

Machel Pantin — ▷ + ▽ + ▷ + Any punch button.

Anja Dass — Wha’s the scene horse?

Megan ‘Chinchilla’ Lee Yuen — For Chinese delights.

Clint ‘Tuffzool’ Sieunarine — My brother of the sword.

Michael Doré — Time for an I.M.O. Q6?

Deepika ‘Special K’ Khurana — For more patience, love and understanding than one has ever deserved.

## Declaration

The author declares that the material contained in this thesis is entirely his own work. As is standard practice, the subject matter developed builds on existing theory, and clear citations and references are provided where necessary.

Although material contained herein may be submitted for publication at a later date, the author has not published any work which forms part of this thesis.

No part of this thesis has been submitted, for the purposes of a degree or otherwise, to any other university or educational institution.

## **Abstract**

A set of fundamental algorithms for computing with polycyclic-by-finite groups is presented here. Polycyclic-by-finite groups arise naturally in a number of contexts; for example, as automorphism groups of large finite soluble groups, as quotients of finitely presented groups, and as extensions of modules by groups.

No existing mode of representation is suitable for these groups, since they will typically not have a convenient faithful permutation representation.

A mixed mode is used to represent elements of such a group; utilising a polycyclic presentation or a power-conjugate presentation for the elements of the normal subgroup, and a permutation representation for the elements of the quotient.

## Notation

$\mathbb{Z}$	the set of integers, $\{\dots, -1, 0, 1, \dots\}$
$\mathbb{Z}^+$	the set of positive integers, $\{1, 2, \dots\}$
$\mathbb{Z}^-$	the set of negative integers, $\{\dots, -2, -1\}$
$\mathbb{Z}_n, \mathbb{Z}/n\mathbb{Z}$	the set of integers modulo $n$
$\mathbb{N}$	the set of natural numbers, $\{0, 1, \dots\}$
$\mathbb{F}_{p^n}$	the finite field of order $p^n$ where $p$ is prime and $n \in \mathbb{Z}^+$
$\text{GL}(d, R)$	the group of invertible $d \times d$ matrices over a ring, $R$
$\text{SL}(d, R)$	the group of $d \times d$ matrices with determinant 1 over a ring, $R$
$\text{PGL}(d, q)$	the projective general linear group of degree $d$ over the field, $\mathbb{F}_q$ , $q$ a prime power
$\text{PSL}(d, q)$	the projective special linear group of degree $d$ over the field, $\mathbb{F}_q$ , $q$ a prime power
$\text{GU}(d, q^2)$	the general unitary group of degree $d$ over the field, $\mathbb{F}_{q^2}$ , $q$ a prime power
$\text{SU}(d, q^2)$	the special unitary group of degree $d$ over the field, $\mathbb{F}_{q^2}$ , $q$ a prime power
$\text{PGU}(d, q^2)$	the projective general unitary group of degree $d$ over the field, $\mathbb{F}_{q^2}$ , $q$ a prime power
$\text{PSU}(d, q^2)$	the projective special unitary group of degree $d$ over the field, $\mathbb{F}_{q^2}$ , $q$ a prime power



---

$\text{AGL}(d, q)$	the group corresponding to the action of $\text{GL}(d, \mathbb{F}_q)$ on the affine points of the $d$ -dimensional vector space over the field, $\mathbb{F}_q$ , $q$ a prime power
$[3 \dots 8]$	the array, $[3, 4, 5, 6, 7, 8]$
$[11 \dots -1 \text{ by } -3]$	the array, $[11, 8, 5, 2, -1]$
$[]$	the empty array

# Chapter 1

## Introduction

This chapter aims to acquaint the reader with the topic and scope of the research undertaken towards the completion of this thesis. Firstly, the class of *polycyclic-by-finite* groups is discussed briefly, after which the objectives and results of the project are outlined.

### 1.1 The Class of Polycyclic-by-Finite Groups

Let  $\mathcal{P}$  and  $\mathcal{Q}$  be properties of groups. A group is said to be *poly- $\mathcal{P}$*  if it admits a finite subnormal series such that every factor group has property  $\mathcal{P}$ . A group,  $G$ , is a  *$\mathcal{P}$ -by- $\mathcal{Q}$  group* if it has a normal subgroup,  $N$ , such that  $N$  has property  $\mathcal{P}$ , and  $G/N$  has property,  $\mathcal{Q}$ .

Thus, a group is *polycyclic-by-finite* if it has a normal polycyclic subgroup of finite index. That is, if it has normal subgroup of finite index that admits a subnormal series with cyclic factors.

Segal (1983) proves that the property “polycyclic-by-finite” is equivalent to the properties:

- (i) poly-(cyclic or finite),
- (ii) (poly- $C_\infty$ )-by-finite,

where  $C_\infty$  is the property of being infinite cyclic.

By a well-known theorem of P. Hall, every polycyclic-by-finite group is finitely presented — and in fact, polycyclic-by-finite groups form the largest known section-closed class of finitely presented groups. It is this fact that makes polycyclic-by-finite groups natural objects of study from the algorithmic standpoint. For an excellent theoretical account of the algorithmic decision theory of polycyclic-by-finite groups, the reader is referred to (Baumslag et al., 1991).

In contrast to the paper of Baumslag et al. (1991), this thesis explores the computational properties of polycyclic-by-finite groups from a practical perspective, attempting to produce algorithms which are conducive to computer implementation.

In this context, the algorithms developed are targeted not only at infinite polycyclic-by-finite groups, but, in fact, primarily at finite insoluble groups with a large soluble normal subgroup, as such groups do not often have a convenient permutation representation. These groups, although trivially polycyclic-by-finite, may be viewed as polycyclic-by-finite with non-trivial polycyclic normal subgroup, leading to a computationally effective representation. Groups of this type arise naturally in many applications such as automorphism groups of large finite soluble groups, as quotients of finitely presented groups, and as extensions of modules by groups.

## 1.2 Overview of Project

The objectives and outcome of the research governing this thesis are outlined in this section.

### 1.2.1 Objectives

The central goal of this thesis is to define a data structure, and develop fundamental, machine implementable algorithms, for the class of polycyclic-by-finite groups. By a data structure, one means a mode by which such groups may be represented on

the computer. Fundamental algorithms for a class of groups include algorithms to perform the tasks of multiplication, element inversion and subgroup generation.

The material developed also aims to use the fundamental algorithms developed to perform more advanced structural computations in polycyclic-by-finite groups such as finding Sylow subgroups (in the finite case), computing centralisers and testing element conjugacy.

Focus was initially restricted to finite insoluble groups, as many of the interesting examples of polycyclic-by-finite groups that arise in practice are in fact finite (see Section 1.1). However, taking into account the recent developments and ongoing research in the area of computing with infinite polycyclic groups (see Eick, 2001), it was deemed prudent to allow for the possibility of infinite groups, and the parameters of the problem were broadened accordingly.

### 1.2.2 Results

This subsection contains a synopsis of the results emerging from this thesis.

#### **Theoretical**

The main body of the thesis begins by defining a normal form for the elements of a polycyclic-by-finite group, after which, a data structure which facilitates multiplication of elements is introduced. This material forms the content of Chapter 3, which culminates with a detailed description of the multiplication algorithm.

The question of subgroup generation is addressed in Chapter 4. In the discussion of this topic, the underlying problem of generating a subgroup is extracted and treated separately, after which, there follows a description of how the data structure is computed for a subgroup. This dissection results in the chapter exhibiting a more pedagogical style. Several applications of subgroup generation are presented in the final section of this chapter.

A selection of advanced structural computations in polycyclic-by-finite groups is discussed in Chapter 5. Specifically, algorithms to compute the centre of the group and the centraliser of an element are presented, along with a constructive method to test element conjugacy. Substantial use of linear algebra and representation theory is made here.

## Practical

In keeping with the “machine implementable” ethos of this course of research, every function described in thesis has been fully implemented using the Magma Computational Algebra System<sup>1</sup>. This suite of functions forms a major portion of a complete package for computing with polycyclic-by-finite groups in Magma. The implementation was done in parallel with the development of the algorithms, and forms a significant part of the project.

An up-to-date version of the source code (written in the Magma language) fulfilling the implementation may be found on the compact disc accompanying this thesis. The reader should consult the documentation provided Appendix A for directions on how to load and use the package. Some example computations are given in Section 6.1 of Chapter 6. Although extensive testing has been done, the implementation is still in developmental stage and, in rare circumstances, bugs may arise, especially when attempting to compute with infinite polycyclic-by-finite groups. Source code listings are given in Appendix B.

---

<sup>1</sup>Magma is a large, well-supported software package designed for computations in algebra, number theory, algebraic geometry and algebraic combinatorics. See <http://magma.maths.usyd.edu.au/magma/> for details.

### 1.2.3 Notes and Assumptions

#### Existing Computational Representations

At points in the thesis, the reader may encounter the phrase “without performing any element arithmetic in  $E$ ”, where  $E$  is a polycyclic-by-finite group. This simply means that, while there exists some (possibly inefficient) representation for  $E$  on the computer, this representation is not used to perform element arithmetic. Indeed, the first and most important goal of the theory is to set up machinery so that elements of  $E$  can be manipulated by performing operations only within the normal polycyclic subgroup and the associated finite quotient, without appealing to any existing representation of  $E$ .

On a related point, while, for a given polycyclic-by-finite group,  $E$ , there may be many different decompositions of  $E$  as an extension of a polycyclic group by a finite group, the algorithms presented in this thesis are targeted at those decompositions in which the finite quotient is relatively small and therefore admits a permutation representation of manageable degree. Thus, it shall hereinafter be assumed that the polycyclic-by-finite groups in this thesis can be so decomposed.

#### Complexity

Time and space complexity analyses are provided for the main algorithms around which the theory is built. The time complexity of an algorithm shall be stated in terms of the number of multiplications performed using the given computational representation of the group in question; in the case of later algorithms, the time cost is assessed by counting the number of multiplications performed using the newly devised representation for the polycyclic-by-finite group.

# Chapter 2

## Background Material

This chapter contains a summary of the required background material, and introduces notation that is used throughout the rest of the thesis.

### 2.1 Permutation Groups

Permutation group algorithms are among the best developed parts of computational group theory. The base and strong generating set data structure and the Schreier–Sims algorithm introduced by Sims (1970, 1971) form the backbone of this area, and the resulting methods enable detailed structural computations to be carried out routinely in permutation groups of degree up to about  $10^7$ .

As discussed in Chapter 1, the chosen representation for polycyclic-by-finite groups involves viewing the finite quotients primarily as permutation groups. Thus, the algorithms developed here for polycyclic-by-finite groups rely on the existing and well established methods for manipulating finite groups of permutations; the fundamental concepts of which are presented in this section. For a detailed account of the material discussed here, the reader is referred to (Holt et al., 2005, Chap. 4) and (Seress, 2003).

### 2.1.1 Definitions and Notation

In the context of permutation groups, the notation used is similar to that of Seress (2003) and Holt et al. (2005). The required definitions are reproduced concisely here.

The cycle notation shall be used for permutations, with the identity permutation denoted by  $()$ . The group of all permutations of an  $n$ -element set,  $\Omega$ , is denoted by  $\text{Sym}(\Omega)$ , or  $S_n$  if only the size of  $\Omega$  is relevant. The image of  $\alpha \in \Omega$  under the permutation  $g \in \text{Sym}(\Omega)$  is written as  $\alpha^g$ . The alternating group on  $\Omega$  is denoted by  $\text{Alt}(\Omega)$  (or  $A_n$  in the case of a generic  $n$ -element set). The *support* of  $g \in \text{Sym}(\Omega)$ , denoted by  $\text{supp}(g)$ , consists of those elements of  $\Omega$  that are actually displaced by  $g$ , and the *degree* of  $g$ ,  $\deg(g)$ , is defined to be the size of this set, i.e.  $\text{supp}(g) = \{\omega \in \Omega \mid \omega^g \neq \omega\}$ ,  $\deg(g) = |\text{supp}(g)|$ . The set of fixed points of  $g$  is defined as  $\text{fix}(g) = \Omega \setminus \text{supp}(g)$ .

A group,  $G$ , is said to *act on* a set,  $\Delta$ , if there exists a homomorphism (called an action),  $\varphi: G \rightarrow \text{Sym}(\Delta)$ , and the *degree* of  $\varphi$  is defined to be  $|\Delta|$  (if  $G \leq \text{Sym}(\Delta)$ , then one speaks of the degree of  $G$ ). The action,  $\varphi$ , is *faithful* if its kernel,  $\ker \varphi$ , is the singleton containing the identity. The image,  $\varphi(G)$ , is denoted by  $G^\Delta$ . In the special case where  $G \leq \text{Sym}(\Omega)$ ,  $\Delta \subseteq \Omega$  is fixed by  $G$ , and  $\varphi$  is the restriction of permutations to  $\Delta$ ,  $G^\Delta$  is also denoted by  $G|_\Delta$ .

Let  $G \leq \text{Sym}(\Omega)$ . The orbit of  $\omega \in \Omega$  under  $G$  is the set of images,  $\omega^G = \{\omega^g \mid g \in G\}$ . For  $\Delta \subseteq \Omega$  and  $g \in G$ ,  $\Delta^g = \{\delta^g \mid \delta \in \Delta\}$ . The group,  $G$ , is said to be *transitive* on  $\Omega$  if it has only one orbit, and  $G$  is  *$k$ -transitive* ( $k \leq n$ ) if the action of  $G$  induced on the set of ordered  $k$ -tuples of distinct elements of  $\Omega$  is transitive. The largest such  $k$  is called the *degree of transitivity* of  $G$ .

If  $G$  is transitive and  $\Delta \subseteq \Omega$ , then  $\Delta$  is called a *block of imprimitivity* for  $G$  if for all  $g \in G$  either  $\Delta^g = \Delta$  or  $\Delta^g \cap \Delta = \emptyset$ . The group,  $G$ , is called *primitive* if all blocks have 0, 1 or  $|\Omega|$  elements. If  $\Delta$  is a block then the set of images of  $\Delta$  is



a partition of  $\Omega$ , which is called a *block system*, and an action of  $G$  is induced on the block system. A block is called *minimal* if it has more than one element and its proper subsets of size at least 2 are not blocks. A block is called *maximal* if the only block properly containing it is  $\Omega$ . A block system is *maximal* if it consists of minimal blocks, whereas a block system is *minimal* if it consists of maximal blocks. The action of  $G$  on a minimal block system is primitive.

The *pointwise* and *setwise stabilisers* of  $\Delta \subseteq \Omega$  are denoted by  $G_{(\Delta)}$  and  $G_\Delta$  respectively. i.e.  $G_{(\Delta)} = \{g \in G \mid (\forall \delta \in \Delta)(\delta^g = \delta)\}$ ,  $G_\Delta = \{g \in G \mid \Delta^g = \Delta\}$ . If  $\Delta$  has only one or two elements, then the set braces and parentheses are often dropped from the notation of the pointwise stabiliser; in particular  $G_\delta$  denotes the stabiliser of  $\delta \in \Omega$ . If  $\Delta = (\delta_1, \dots, \delta_m)$  is a *sequence* of elements of  $\Omega$ , then  $G_\Delta$  denotes the pointwise stabiliser of that sequence.

The group,  $G$ , is said to be *semiregular* if  $G_\delta = 1$  for all  $\delta \in \Omega$ , and  $G$  is *regular* if it is transitive and semiregular. The group,  $G$ , is a *Frobenius group* if it is transitive, not regular, but for which  $G_{\alpha,\beta} = 1$  for distinct  $\alpha, \beta \in \Omega$ .

If  $g \in \text{Sym}(\Omega)$  then a bijection  $\phi: \Omega \rightarrow \Delta$  naturally defines a permutation  $\bar{\phi}(g) \in \text{Sym}(\Delta)$  by the rule  $\phi(\omega)^{\bar{\phi}(g)} = \phi(\omega^g)$  for all  $\omega \in \Omega$ . The groups,  $G \leq \text{Sym}(\Omega)$  and  $H \leq \text{Sym}(\Delta)$ , are *permutation isomorphic*, written  $H \sim G$ , if there is a bijection,  $\phi: \Omega \rightarrow \Delta$ , such that  $\bar{\phi}(G) = \{\bar{\phi}(g) \mid g \in G\} = H$ .

### 2.1.2 Computation of Orbits and Stabilisers

In algorithms involving permutation groups, calculations of point orbits and transversals of the associated stabilisers are performed frequently.

#### The Basic Orbit Algorithm

Let  $G = \langle X \rangle$  be a finite group acting on a finite set,  $\Omega$ , and let  $\alpha \in \Omega$ . In a computational context, the orbit,  $\alpha^G$ , may be viewed as the vertex set of a breadth-

first-search tree rooted at  $\alpha$  in the directed graph,  $D(\Omega, \vec{E})$ , with vertex set  $\Omega$  and edge set  $\vec{E} = \{(\overrightarrow{\beta, \gamma}) \mid \beta, \gamma \in \Omega \wedge (\exists x \in X \cup X^{-1})(\beta^x = \gamma)\}$ . Assuming that the images,  $\beta^x$ , can be computed, and that points of  $\Omega$  can be compared, a standard breadth-first-search method may be employed to find this vertex set along with a transversal of  $G_\alpha$  in  $G$ .

The algorithm starts with  $L_0 = \{\alpha\}$  and computes the sets,  $L_i$ , according to the recursive definition

$$L_i = \{\gamma \in \Omega \mid (\exists \beta \in L_{i-1})(\overrightarrow{\beta, \gamma} \in \vec{E})\} \setminus \bigcup_{j < i} L_j,$$

terminating when  $L_m = \emptyset$  for some  $m$ . The finiteness of  $\Omega$  guarantees that the algorithm does in fact terminate. The  $L_i$  are called the *levels* of the breadth-first-search tree.

The set,  $\Delta = \bigcup_{j < m} L_j$ , is the required orbit; the containment  $\Delta \subseteq \alpha^G$  clearly holds, and the reverse inclusion follows from the fact that, by construction,  $\Delta^x = \Delta$  for all  $x \in X$ , hence  $\Delta^g = \Delta$  for all  $g \in G$ . A transversal of  $G_\alpha$  in  $G$  is found by keeping a record, at each level, the permutations of  $X \cup X^{-1}$  that are used to compute the points that lie in the subsequent level.

Assuming that the tasks of finding an image,  $\beta^x$ , and testing element membership in subsets of  $\Omega$  can both be performed in constant time, the orbit computation method presented here has time complexity  $O(|\Delta||X|)$ .

## Schreier Vectors

Computing and storing a stabiliser transversal explicitly may require  $\Theta(n^2)$  memory. This approach becomes impractical when dealing with permutation groups of large degree. The Schreier vector data structure offers an improvement on this space complexity.

**Definition 2.1.** Let  $G = \langle X \rangle$  be a finite permutation group acting on a finite set,

$\Omega$ , and let  $\alpha \in \Omega$ . A *Schreier vector* (or *Schreier tree*) for  $\alpha$  relative to  $X$  is a directed labelled tree,  $T$ , with all edges directed to the root,  $\alpha$ , and edge labels taken from the set,  $X \cup X^{-1}$ . The vertices of  $T$  are the points of the orbit,  $\alpha^G$ , and the edge labels are such that if  $(\overrightarrow{\gamma, \delta})$  is an edge with label  $x$  then  $\gamma^x = \delta$ .

Let  $G$  be as described in Definition 2.1, let  $|\Omega| = n$  and let  $X = \{x_1, \dots, x_r\}$ . Let  $T$  be a Schreier vector for  $\alpha$  relative to  $X$ . If  $\gamma$  is a vertex of  $T$  then the sequence of edge labels along the unique path from  $\gamma$  to  $\alpha$  is a word in the elements of  $X \cup X^{-1}$  which, when viewed as a product of permutations, moves  $\gamma$  to  $\alpha$ . Thus, Schreier vectors define inverses of a set of coset representatives for  $G_\alpha$  in  $G$ .

The Schreier vector data structure is usually implemented using two arrays of length equal to the size of the orbit in question. For the group,  $G$ , and the point,  $\alpha$ , the first array,  $\Delta$ , holds the elements of  $\alpha^G$ , with  $\Delta[1] = \alpha$ . The second,  $v$ , is an integer array whose entries satisfy:

$$(i) \quad v[1] = 0.$$

(ii) For  $j = 2, \dots, |\alpha^G|$ , let  $e_j$  be the edge of  $T$  with first component  $\Delta[j]$ , then

$$v[j] = \begin{cases} i & \text{if } e_j \text{ has label } x_i, \\ -i & \text{if } e_j \text{ has label } x_i^{-1}. \end{cases}$$

The maximum length of an orbit is  $n$ , hence, storing a Schreier vector in the manner described above requires only  $O(n)$  memory. There is a time-space trade-off associated with Schreier vectors, for, reconstructing a coset representative from a Schreier vector (by retracing a sequence of edge labels as described above) requires  $O(n)$  multiplications in the permutation group, in contrast to the constant time required when the transversals are stored explicitly.

### 2.1.3 Bases and Strong Generating Sets

The notion of a base and strong generating set of a permutation group is fundamental in the theory. The resulting data structure allows for efficient manipulation of permutation groups.

#### Basic Definitions

Let  $\Omega = \{1, \dots, n\}$  and  $G \leq \text{Sym}(\Omega)$ . A sequence,  $B = (\beta_1, \dots, \beta_l)$ , of elements belonging to  $\Omega$  is called a *base* for  $G$  if the only element of  $G$  to fix  $B$  pointwise is the identity. The sequence  $B$  defines a stabiliser subgroup chain,

$$G = G^{[1]} \geq G^{[2]} \geq \dots \geq G^{[l]} \geq G^{[l+1]} = 1, \quad (2.1)$$

where  $G^{[i]} = G_{(\beta_1, \dots, \beta_{i-1})}$  ( $i > 1$ ) is the pointwise stabiliser of  $\{\beta_1, \dots, \beta_{i-1}\}$ . The base,  $B$ , is called *non-redundant* if  $G^{[i+1]} < G^{[i]}$  for all  $i = 1, \dots, l$ . The orbits,  $\beta_i^{G^{[i]}}$ , are called the *basic orbits* or *fundamental orbits* of  $G$  (relative to  $B$ ).

By repeated applications of Lagrange's Theorem and the Orbit-Stabiliser Theorem, one obtains

$$|G| = \prod_{i=1}^l [G^{[i]} : G^{[i+1]}] = \prod_{i=1}^l |\beta_i^{G^{[i]}}|. \quad (2.2)$$

Now, for each  $i$ , clearly  $|\beta_i^{G^{[i]}}| \leq n$ . Moreover, if  $B$  is non-redundant then  $|\beta_i^{G^{[i]}}| \geq 2$ .

These inequalities, combined with Equation (2.2) immediately yield

$$2^{|B|} \leq |G| \leq n^{|B|},$$

or

$$\frac{\log_2 |G|}{\log_2 n} \leq |B| \leq \log_2 |G|.$$

That is, the length of a base is logarithmic in the size of the permutation group. This inequality is used frequently in the complexity calculations of this, and subsequent

chapters.

A *strong generating set* for  $G$  relative to  $B$  is a generating set,  $S$ , for  $G$  with the property that

$$\langle S \cap G^{[i]} \rangle = G^{[i]} \quad (2.3)$$

for  $i = 1, \dots, l + 1$ .

The *base image* of an element  $g \in G$  is the sequence  $B^g = (\beta_1^g, \dots, \beta_l^g)$ .

**Observation 2.2.** The base image of  $g$  uniquely determines the element,  $g$ . To see this, suppose that  $B^g = B^h$  for some  $h \in G$ . Then  $B^{gh^{-1}} = B$  whence  $gh^{-1} = 1$  by the definition of a base.

### The Sifting Procedure

Given a base for a permutation group, one may define a normal form for group elements relative to this base.

**Proposition 2.3.** *Let  $U_i$  be a right transversal of  $G^{[i+1]}$  in  $G^{[i]}$  for  $i = 1, \dots, l$ . Then, every element  $g \in G$  may be expressed uniquely as*

$$g = u_l u_{l-1} \cdots u_1$$

where  $u_i \in U_i$ .

*Proof.* Induction on the length of a base may be used to prove the existence of the asserted decomposition. The case  $l = 1$  is trivial, for the required decomposition is simply the element itself.

Assume that the decomposition exists for bases of length  $l - 1$ , and let  $g \in G$ . The Orbit–Stabiliser Theorem provides an element,  $u_1 \in U_1$ , such that  $\beta_1^g = \beta_1^{u_1}$ . The element,  $gu_1^{-1}$ , belongs to the group  $G^{[2]}$  which has base,  $(\beta_2, \dots, \beta_l)$  of length  $l - 1$ , and right transversals,  $U_i$ , for  $i = 2, \dots, l$ . By the inductive hypothesis,  $gu_1^{-1}$

has decomposition  $u_l u_{l-1} \cdots u_2$ , where  $u_i \in U_i$ . This gives

$$g = u_l u_{l-1} \cdots u_1$$

as required.

To prove uniqueness, assume that  $u_l u_{l-1} \cdots u_1$  and  $u'_l u'_{l-1} \cdots u'_1$  are different decompositions of an element,  $g \in G$ . Let  $j$  be the smallest index such that  $u_j \neq u'_j$ . Then, by the Orbit-Stabiliser Theorem,  $\beta_j^{u_j} \neq \beta_j^{u'_j}$ , whence  $u_l u_{l-1} \cdots u_1$  and  $u'_l u'_{l-1} \cdots u'_1$  give rise to different base images, contradicting Observation 2.2. It follows that  $u_i = u'_i$  for each  $i$ .  $\square$

By Proposition 2.3, the transversals,  $U_i$ , provide a convenient normal form for elements of  $G$ .

The decomposition in the statement of the proposition can be done algorithmically as follows. Given  $g \in G$ , find the coset representative,  $u_1 \in U_1$ , such that  $\beta_1^g = \beta_1^{u_1}$  and compute  $g_2 = g u_1^{-1} \in G^{[2]}$ . Then find  $u_2 \in U_2$  such that  $\beta_2^{g_2} = \beta_2^{u_2}$  and compute  $g_3 = g_2 u_2^{-1}$ . Iterate  $l$  times to obtain the required factorisation. This procedure is called *sifting* or *stripping*. Sifting involves constructing exactly one transversal element for each base point, and thus requires  $O(n \log_2 |G|)$  multiplications in the permutation group.

Sifting can also be used to test membership in  $G$ . Given  $h \in \text{Sym}(\Omega)$ , one attempts to factor  $h$  as a product of coset representatives,  $u_i$ . If the factorisation is successful then  $h \in G$ . Two things may go awry; it is possible that for some  $i \leq l$ , the ratio  $h_i = h u_1^{-1} u_2^{-1} \cdots u_{i-1}^{-1}$  computed by the sifting procedure carries  $\beta_i$  out of the orbit  $\beta_i^{G^{[i]}}$ , or  $h_{l+1} = h u_1^{-1} u_2^{-1} \cdots u_l^{-1} \neq 1$ . In either case,  $h \notin G$ . The last ratio  $h_i$  computed by the sifting procedure is called the *siftee* of  $h$ .

### The Schreier–Sims Algorithm

The Schreier–Sims algorithm is used to construct a base and strong generating set for a given permutation group. The method is based on the following lemmas, taken from (Seress, 2003, Chap. 4).

**Lemma 2.4** (Schreier). *Let  $H \leq G = \langle S \rangle$  and let  $R$  be a right transversal of  $H$  in  $G$  with  $1 \in R$ . For  $g \in G$ , denote the unique element of  $Hg \cap R$  by  $\bar{g}$ . Then the set,*

$$T = \{rs(\overline{rs})^{-1} \mid r \in R, s \in S\},$$

*generates  $H$ .*

*Proof.* By definition, the elements of  $T$  are in  $H$ , so it suffices to show that  $T \cup T^{-1}$  generates  $H$ . Note that  $T^{-1} = \{rs(\overline{rs})^{-1} \mid r \in R, s \in S^{-1}\}$ .

Let  $h \in H$  be arbitrary. Since  $H \leq G$ ,  $h$  can be written in the form  $h = s_1 \cdots s_k$  for some non-negative integer,  $k$ , where  $s_i \in S \cup S^{-1}$  for each  $i$ . Define a sequence  $h_0, h_1, \dots, h_k$  of group elements such that

$$h_j = t_1 \cdots t_j r_{j+1} s_{j+1} \cdots s_k \quad (2.4)$$

with  $t_i \in T \cup T^{-1}$  for each  $i \leq j$ ,  $r_{j+1} \in R$ , and  $h_j = h$ . Let  $h_0 = 1s_1 \cdots s_k$ . Recursively, if  $h_j$  is already defined, then let  $t_{j+1} = r_{j+1}s_{j+1}(\overline{r_{j+1}s_{j+1}})^{-1}$  and  $r_{j+2} = \overline{r_{j+1}s_{j+1}}$ . Clearly,  $h_{j+1} = h_j = h$ , and  $h_{j+1}$  has the form of Equation (2.4) as required.

Thus,  $h = h_k = t_1 \cdots t_k r_{k+1}$ . Since  $h \in H$  and  $t_1 \cdots t_k \in \langle T \rangle \leq H$ , it follows that  $r_{k+1} \in H \cap R = \{1\}$ , whence  $h \in \langle T \rangle$ . Thus  $H \leq \langle T \rangle$ .  $\square$

The elements of the set,  $T$ , in Lemma 2.4 are called *Schreier generators* for the subgroup  $H$ .

**Lemma 2.5** (Sims, 1970). *Let  $G \leq \text{Sym}(\Omega)$  and  $\{\beta_1, \dots, \beta_l\} \subseteq \Omega$ . For each  $j$  in  $\{1, \dots, l+1\}$ , let  $S_j \subseteq G_{(\beta_1, \dots, \beta_{j-1})}$  such that  $\langle S_j \rangle \geq \langle S_{j+1} \rangle$  holds for  $j \leq l$ . If  $G = \langle S_1 \rangle$ ,  $S_{l+1} = \emptyset$ , and*

$$\langle S_j \rangle_{\beta_j} = \langle S_{j+1} \rangle \quad (2.5)$$

*holds for each  $j$ , then  $B = (\beta_1, \dots, \beta_l)$  is a base for  $G$  and  $S = \bigcup_{j=1}^l S_j$  is a strong generating set for  $G$  relative to  $B$ .*

*Proof.* Induction on  $l$  is used here. The case  $l = 1$  is trivial, for  $\bigcup_{j=1}^1 S_j$  clearly fulfils the requirements of a strong generating set (relative to the base containing the single point  $\beta_1$ ) for the group,  $\langle S_1 \rangle$ .

Assume that the result holds for bases of length  $l - 1$ . Then, in particular,  $S^* = \bigcup_{j=2}^l S_j$  is a strong generating set for  $\langle S_2 \rangle$  relative to the base  $B^* = (\beta_2, \dots, \beta_l)$ .

Let  $G^{[i]} = G_{(\beta_1, \dots, \beta_{i-1})}$  for  $i = 2, \dots, l+1$ . To prove the lemma, it is required to verify that Equation (2.3) holds for each  $i$ . Setting  $j = 1$  in Equation (2.5) yields

$$G_{\beta_1} = \langle S_1 \rangle_{\beta_1} = \langle S_2 \rangle,$$

which implies the containment,

$$G_{\beta_1} \leq \langle S \cap G_{\beta_1} \rangle.$$

The reverse inclusion is obvious, thus

$$\langle S \cap G^{[2]} \rangle = G^{[2]},$$

and Equation (2.3) is satisfied for  $i = 2$ .

For  $i > 2$ , the inductive hypothesis implies that  $S^* \cap G_{(\beta_1, \dots, \beta_{i-1})}$  generates  $\langle S_2 \rangle_{(\beta_1, \dots, \beta_{i-1})}$ , and so  $G^{[i]} \geq \langle S \cap G_{(\beta_1, \dots, \beta_{i-1})} \rangle \geq \langle S^* \cap G_{(\beta_1, \dots, \beta_{i-1})} \rangle = \langle S_2 \rangle_{(\beta_1, \dots, \beta_{i-1})} =$



$(G_{\beta_1})_{(\beta_1, \dots, \beta_{i-1})} = G^{[i]}$ . Therefore,

$$\langle S \cap G^{[i]} \rangle = G^{[i]}$$

for each  $i$ , as required.  $\square$

Given a permutation group,  $G = \langle T \rangle$ , acting on a set,  $\Omega$ , a base and strong generating set can be constructed in the following way. A data structure containing a list,  $B = (\beta_1, \dots, \beta_k)$ , of already known elements of a non-redundant base is maintained, along with an approximation,  $S_i$ , for a generator set of the stabiliser,  $G_{(\beta_1, \dots, \beta_{i-1})}$ , for each  $i \in \{1, \dots, k\}$ . Throughout execution, the  $S_i$  satisfy the property that, for all  $i$ ,  $\langle S_i \rangle \geq \langle S_{i+1} \rangle$ . The data structure is said to be *up-to-date below level  $j$*  if Equation (2.5) holds for each  $i$  in the range  $j < i \leq k$ .

In the case where the data structure is up-to-date below level  $j$ , a transversal,  $R_j$ , of  $\langle S_j \rangle_{\beta_j}$  in  $\langle S_j \rangle$  is computed. Then a check is made to determine whether Equation (2.5) is satisfied for  $i = j$ . By Lemma 2.4, this can be done by sifting the Schreier generators obtained from  $R_j$  and  $S_j$  in the group,  $\langle S_{j+1} \rangle$ . (In this group, membership testing is possible, since Lemma 2.5 implies that  $\bigcup_{i=j+1}^k S_i$  is a strong generating set for  $\langle S_{j+1} \rangle$ .) If all Schreier generators are in  $\langle S_{j+1} \rangle$  then the data structure is up-to-date below level  $j - 1$ ; otherwise a non-trivial siftee,  $h$ , at level  $t$  for some  $t$  in the range,  $j + 1 \leq t \leq k + 1$ , is added to  $S_t$ , and the data structure is now up-to-date below level  $t$ . If  $t = k + 1$ , a new base point,  $\beta_{k+1}$ , is chosen from  $\text{supp}(h)$ .

The algorithm initialises  $B$  to contain a single point,  $\beta_1 \in \Omega$ , that is moved by at least one generator in  $T$  and sets  $S_1$  to  $T$ . At that moment, the data structure is up-to-date below level 1; the algorithm terminates when the data structure becomes up-to-date below level 0. Lemma 2.5 immediately implies correctness.

Bounds on the time-space requirements of the algorithm are given in Theorem 2.6. For a proof and a detailed discussion, the reader is referred to (Seress,

2003, Chap. 4).

**Theorem 2.6.** *Given a finite permutation group  $G = \langle X \rangle$  acting on a set of cardinality  $n$ , a base and strong generating set for  $G$  can be computed by deterministic algorithms in  $O((n \log_2 |G|)^3 + |X|n^3 \log_2 |G|)$  time using  $O(n(\log_2 |G|)^2 + |X|n)$  memory.*

## 2.2 Polycyclic Groups

A group,  $G$ , is said to be *polycyclic* if it has a descending chain of subgroups

$$G = G_1 \supseteq \cdots \supseteq G_r \supseteq G_{r+1} = 1,$$

in which  $G_i/G_{i+1}$  is cyclic for  $i = 1, 2, \dots, r$ . Such a chain of subgroups is called a *polycyclic series*.

If  $G_i/G_{i+1} = \langle x_i G_{i+1} \rangle$  for each  $i$ , then  $G = \langle x_1, \dots, x_r \rangle$ . Thus, every polycyclic group is finitely generated. The sequence,  $(x_1, \dots, x_r)$ , is called a *polycyclic generating sequence* for  $G$ .

Subgroups and quotients of polycyclic groups are themselves polycyclic, for if  $H \leq G$  and  $N \trianglelefteq G$ , then

$$H = H \cap G_1 \supseteq \cdots \supseteq H \cap G_r \supseteq H \cap G_{r+1} = 1$$

and

$$G/N = G_1N/N \supseteq \cdots \supseteq G_rN/N \supseteq G_{r+1}N/N = 1,$$

are polycyclic series for  $H$  and  $G/N$  respectively,

Lemma 2.8 gives a characterisation of polycyclic groups as a subclass of the soluble groups. First, a necessary and sufficient condition for an abelian group to be polycyclic:

**Lemma 2.7.** *An abelian group is polycyclic if and only if it is finitely generated.*

*Proof.* One direction is clear, for every polycyclic group is finitely generated, and so, in particular, every abelian polycyclic group is finitely generated.

Conversely, suppose that  $A = \langle a_1, \dots, a_r \rangle$  is abelian. Then

$$A = \langle a_1, \dots, a_r \rangle \supseteq \langle a_1, \dots, a_{r-1} \rangle \cdots \supseteq \langle a_1, a_2 \rangle \supseteq \langle a_1 \rangle \supseteq 1$$

is a series with cyclic factors, showing  $A$  to be polycyclic.  $\square$

**Lemma 2.8.** *The polycyclic groups are exactly the soluble groups for which every subgroup is finitely generated.*

*Proof.* Suppose that  $G$  is polycyclic. Then, it follows immediately from the definition of a polycyclic group that  $G$  has a subnormal series with abelian factors, and is hence soluble. If  $H \leq G$ , then  $H$  is polycyclic and hence finitely generated.

Conversely, suppose that  $G$  is a soluble group for which every subgroup is finitely generated. Let

$$G = G_1 \supseteq \cdots \supseteq G_r \supseteq G_{r+1} = 1,$$

be a soluble series for  $G$ . By hypothesis, the abelian factors,  $G_i/G_{i+1}$ , are finitely generated, and hence polycyclic by Lemma 2.7. Thus, the soluble series above may be refined to obtain a polycyclic series for  $G$ , by inserting isomorphic copies of a polycyclic series for each factor,  $G_i/G_{i+1}$ .  $\square$

In contrast to Lemma 2.7, not every finitely generated soluble group is polycyclic. A counterexample is constructed here. Take

$$A = \prod_{i=-\infty}^{\infty} \langle a_i \rangle$$

to be the restricted direct product of infinitely many infinite cyclic groups, and let

$x$  be the automorphism of  $A$  defined by

$$a_i^x = a_{i+1} \quad (-\infty < i < \infty).$$

Then  $x$  has infinite order and generates a group,  $\langle x \rangle \cong \mathbb{Z}$ , of automorphisms of  $A$ . Now form the semidirect product (see Definition 2.10),

$$G = A \rtimes \langle x \rangle.$$

Then  $G$  is soluble, and  $G$  is generated by two elements, namely  $a_0$  and  $x$ . But  $G$  is not polycyclic as the subgroup,  $A$ , of  $G$  is not finitely generated. The group,  $G$ , is the so-called *wreath product*,

$$G = \langle a_0 \rangle \wr \langle x \rangle \cong \mathbb{Z} \wr \mathbb{Z}.$$

Every polycyclic group admits a specific type of finite presentation that allows for efficient structural computation within the group. Finite presentations for polycyclic groups are discussed in the subsections below. Segal (1983) provides an excellent treatise on the beautiful theory of polycyclic groups.

### 2.2.1 Finite Soluble Groups

It is an easy corollary of Lemma 2.8 that, in the finite case, the properties “polycyclic” and “soluble” are equivalent. To prove this directly, argue as follows. Observe that a soluble series for a finite group has finite abelian factors. Therefore, the Basis Theorem for finite abelian groups may then be applied to decompose the finite abelian factors into direct sums of cyclic groups, thereby yielding a refinement of the given soluble series with cyclic factors as required. Conversely, polycyclicity implies solubility.

Thus, every finite soluble group has a subnormal series with cyclic factors. Such a series gives rise to various finite presentations reflecting the polycyclic structure of the group. These presentations are useful because the Word Problem in such presentations can be solved in an algorithmic fashion.

Let  $G$  be a finite soluble group. A presentation for  $G$  of the form,

$$\langle a_1, \dots, a_r \mid a_j^{p_j} = w_{j,j} \text{ for } 1 \leq j \leq r, \\ a_j^{a_i} = w_{i,j} \text{ for } 1 \leq i < j \leq r \rangle,$$

where

(i)  $p_j$  is the least prime such that  $a_j^{p_j} \in \langle a_{j+1}, \dots, a_r \rangle$  for  $j < r$ , and  $a_r^{p_r}$  is the identity, and

(ii)  $w_{i,j}$  is a word in the generators  $a_{i+1}, \dots, a_r$ ,

shall be called a *power-conjugate presentation* for  $G$ . The generators of  $G$  corresponding to  $a_1, \dots, a_r$  in this presentation are known as a *power-conjugate generating sequence* for  $G$ . The relations of the first type are called *power relations*, while those of the second type are called *conjugate relations*.

Let  $G_i = \langle a_i, \dots, a_r \rangle$  for each  $i \leq r$ , and define  $G_{r+1}$  to be the trivial group. The presentation above is said to be *consistent* if  $|G_i/G_{i+1}| = p_i$  for each  $i$ . In this case, every element of  $G$  can be written uniquely in the *normal form*  $a_1^{\alpha_1} \cdots a_r^{\alpha_r}$ , where  $0 \leq \alpha_i < p_i$  for  $i = 1, \dots, r$ .

It is straightforward to show that every finite soluble group possesses a consistent power-conjugate presentation, and conversely, that every power-conjugate presentation defines a finite soluble group. Given a consistent power-conjugate presentation for a group, there exists an algorithm (the *collection algorithm*), which, when given an arbitrary word over the power-conjugate generating sequence, determines the corresponding normal word. In particular, collection can be used to

compute the normal word which is equal to the product of two given normal words, thus implementing the group multiplication.

Power-conjugate presentations are an effective way of representing finite soluble groups, and, over the past two decades, a considerable body of efficient algorithms has been developed for computing with soluble groups defined in terms of power-conjugate presentations. For a survey of the algorithms currently in use for power-conjugate presentations, the reader is referred to (Holt et al., 2005, Chap. 8), and, for a discussion of computation in soluble permutation groups, see (Seress, 2003, Chap. 7).

### 2.2.2 Infinite Polycyclic Groups

A generalisation of the power-conjugate presentation is used to represent infinite polycyclic groups.

Let  $G$  be a polycyclic group. A presentation for  $G$  of the form,

$$\begin{aligned} \langle a_1, \dots, a_r \mid & a_i^{m_i} = w_{i,i} \quad \text{for } i \in I, \\ & a_j^{a_i} = w_{i,j} \quad \text{for } 1 \leq i < j \leq r, \\ & a_j^{a_i^{-1}} = w_{-i,j} \quad \text{for } 1 \leq i < j \leq r, i \notin I \rangle, \end{aligned}$$

where

- (i)  $I \subseteq \{1, \dots, r\}$ ,
- (ii)  $m_i > 1$  for  $i \in I$ , and
- (iii)  $w_{i,j}$  is of the form  $w_{i,j} = a_{|i|+1}^{\epsilon(i,j,|i|+1)} \dots a_r^{\epsilon(i,j,r)}$ , with  $0 \leq \epsilon(i,j,k) < m_k$  if  $k \in I$ .

shall be called a *polycyclic presentation* for  $G$ . The generators of  $G$  corresponding to  $a_1, \dots, a_r$  in this presentation are known as a *polycyclic generating sequence* for  $G$ , and the values,  $m_i$  ( $i \in I$ ), are called the corresponding *polycyclic exponents*. The

relations of the first type are called *power relations*, while those of the second and third types are called *conjugate relations*.

Let  $G_i = \langle a_i, \dots, a_r \rangle$  for each  $i \leq r$ , and define  $G_{r+1}$  to be the trivial group. The presentation above is said to be *consistent* if the quotient,  $G_i/G_{i+1}$ , has order  $m_i$  whenever  $i \in I$ , and is infinite whenever  $i \notin I$ . In this case, every element of  $G$  can be written uniquely in the *normal form*  $a_1^{\alpha_1} \cdots a_r^{\alpha_r}$ , where  $0 \leq \alpha_i < m_i$  for  $i \in I$ .

It is straightforward to show that every polycyclic group possesses a consistent polycyclic presentation, and conversely, that every polycyclic presentation defines a polycyclic group. Given a consistent polycyclic presentation for a group, there exists a version of the collection algorithm, which, when given an arbitrary word over the polycyclic generating sequence, determines the corresponding normal word. In particular, as in the case of power-conjugate presentations, collection can be used to compute the normal word which is equal to the product of two given normal words, thus implementing the group multiplication.

Computing with infinite polycyclic groups are a comparatively new topic in computational group theory and the number of available algorithms is much smaller than in the case of finite polycyclic groups. For an accessible introduction to the algorithmic theory of polycyclic groups, the reader is referred to (Sims, 1994, Chap. 9). A practical account of computing with polycyclic groups can be found in (Eick, 2001).

## 2.3 Representation Theory and Extensions

This section concerns group representation theory, together with the basic theory of extensions of abelian groups. A survey similar to that of Holt et al. (2005) is provided, focusing on the definitions and results needed for later chapters. James and Liebeck (2001) provide an accessible introductory account of Representation Theory, while the reader may consult (Rotman, 2002) for proofs of the more advanced results, in particular those involving representations over finite fields. For a detailed

description of the theory of extensions and cohomology, the reader is referred to Chapter 7 of (Rotman, 1995).

Computation with group representations is a significant subtopic within computational group theory. Some of the methods in this area, particularly those related to group representation theory for its own sake, involve advanced theory. Even for computations that are concerned only with the group-theoretical structure of finite groups, some of the more sophisticated algorithms require some familiarity with representation theory.

The basic reason for this is that if a finite group,  $G$ , has normal subgroups,  $N < M$ , for which  $M/N$  is an elementary abelian  $p$ -group for some prime,  $p$ , then the conjugation action of  $G$  on  $M$  gives rise to a representation of  $G/M$  over the field of order  $p$ , and properties of that representation translate into group-theoretical properties of  $G$ . For example, the representation is irreducible if and only if  $M/N$  is a chief factor of  $G$ .

### 2.3.1 The Terminology of Representation Theory

Let  $K$  be a commutative ring with unity, and let  $G$  be a finite group. The *group ring*,  $KG$ , of  $G$  over  $K$  is defined to be the ring of finite formal sums,

$$\left\{ \sum_{g \in G} r_g g \mid r_g \in K \right\},$$

with the obvious addition and multiplication inherited from that of  $G$ . In fact  $KG$  is an associative algebra with unity, and thus it is a ring with unity and a module over  $K$ . The group ring,  $KG$ , is also known as the *group algebra* of  $G$  over  $K$ .

Let  $M$  be a right (unital)  $KG$ -module. The module product of  $m \in M$  and  $x \in KG$  shall be denoted by  $m \cdot x$ , but when  $x \in K$ , and  $M$  is viewed primarily as a  $K$ -module, then the product may be written as  $xm$ . The commutativity of  $K$  ensures that this causes no problems. From the module axioms, and the fact



that  $(m \cdot g) \cdot g^{-1} = m$  for  $m \in M$ ,  $g \in G$ , one observes that multiplication by a group element  $g \in G$  defines an automorphism of  $M$  as a  $K$ -module. Therefore there is an associated action  $\varphi: G \rightarrow \text{Aut}_K(M)$ , and group action notation,  $m^g$ , shall sometimes be used as alternative to  $m \cdot g$ . Conversely, if  $M$  is a  $K$ -module, then any action  $\varphi: G \rightarrow \text{Aut}_K(M)$  can be used to make  $M$  into a  $KG$ -module.

It shall always be assumed that  $M$  is finitely generated and free as a  $K$ -module, and so, after fixing on a free basis of  $M$ , one may identify  $M$  with  $K^d$  for some  $d$ . Then, using the same free basis of  $M$ ,  $\text{Aut}_K(M)$  may be identified with the group  $\text{GL}(d, K)$ . So the action homomorphism,  $\varphi$ , is  $\varphi: G \rightarrow \text{GL}(d, K)$ , which is the standard definition of a representation of  $G$  of degree  $d$  over  $K$ .

According to basic results from representation theory, two  $KG$ -modules are isomorphic if and only if the associated representations,  $\varphi_1$ ,  $\varphi_2$ , are *equivalent*, which means that they have the same degree and there exists  $\alpha \in \text{GL}(d, K)$  with  $\alpha \cdot \varphi_2(g) = \varphi_1(g) \cdot \alpha$  for all  $g \in G$ .

### 2.3.2 Semidirect Products, Complements, Derivations and First Cohomology Groups

Recall the following definitions.

**Definition 2.9.** An extension,  $G$ , of a group,  $N$ , is called a *split extension* if there is a subgroup,  $C$ , of  $G$  with  $NC = G$  and  $N \cap C = 1$ . Here  $C$  is called a *complement* of  $N$  in  $G$ .

**Definition 2.10.** Let  $G$  and  $M$  be groups, and suppose that there is a given homomorphism,  $\varphi: G \rightarrow \text{Aut}(M)$ . For  $g \in G$ ,  $m \in M$ , abbreviate  $m^{\varphi(g)}$  to  $m^g$ . The *semidirect product* of  $M$  by  $G$  using  $\varphi$ , denoted by  $G \ltimes M$  or  $G \ltimes_{\varphi} M$ , is the set,  $G \times M$ , endowed with the multiplication,  $(g_1, m_1)(g_2, m_2) = (g_1 g_2, m_1^{g_2} m_2)$ , for  $g_1, g_2 \in G$ ,  $m_1, m_2 \in M$ .

**Proposition 2.11.** *Any split extension,  $E$ , of a group,  $M$ , by a group,  $G$ , is isomorphic to the semidirect product  $G \ltimes_{\varphi} M$ , where the action,  $\varphi$ , of  $G$  on  $M$  is defined by the conjugation action of a complement of  $M$  in  $E$  on  $M$ .*

*Proof.* Observe that the semidirect product,  $G \ltimes M$ , is an extension of  $M$  by  $G$ , using the maps,  $M \rightarrow G \ltimes M$  and  $G \ltimes M \rightarrow G$ , defined by  $m \mapsto (1_G, m)$  and  $(g, m) \mapsto g$  respectively. It is a split extension, with complement,  $\{(g, 1_M) \mid g \in G\} \cong G$ .

Now, if a group,  $E$ , has a normal subgroup,  $M$ , with a complement,  $G$  then any  $e \in E$  can be written uniquely as  $e = gm$  where  $g \in G$ ,  $m \in M$ , and

$$g_1 m_1 \cdot g_2 m_2 = g_1 g_2 m_1^{g_2} m_2$$

for  $g_1, g_2 \in G$ ,  $m_1, m_2 \in M$ .

Denote the conjugation action of  $G$  on  $M$  by  $\varphi: G \rightarrow \text{Aut}(M)$ . Then  $E$  is isomorphic to  $G \ltimes_{\varphi} M$  via the map  $gm \mapsto (g, m)$ .  $\square$

In general, different complements could give rise to different actions,  $\varphi$ . However, if  $M$  is abelian, then the actions coming from different complements are the same. It shall be assumed for the remainder of this subsection that  $M$  is abelian, and additive notation shall be employed where appropriate. It shall also be assumed that  $M$  is a  $K$ -module for some commutative ring  $K$  with unity. There is no loss of generality here, because any abelian group can be regarded as a  $\mathbb{Z}$ -module in the obvious manner. In the case where  $M$  is an elementary abelian  $p$ -group for some prime,  $p$ ,  $K$  is taken to be the field  $\mathbb{F}_p$ .

As discussed in Subsection 2.3.1, an action,  $\varphi: G \rightarrow \text{Aut}_K(M)$ , of  $G$  on the  $K$ -module,  $M$ , corresponds to endowing  $M$  with the structure of a  $KG$ -module, and so one may speak about the semidirect product  $G \ltimes M = G \ltimes_{\varphi} M$  of the  $KG$ -module,  $M$ , with  $G$ . The multiplication rule in  $G \ltimes M$ , using additive notation in  $M$ , is

$$(g_1, m_1)(g_2, m_2) = (g_1 g_2, m_1^{g_2} + m_2).$$

A left transversal in  $G \ltimes M$  of the subgroup  $\hat{M} = \{(1_G, m) \mid m \in M\}$  isomorphic to  $M$  has the form  $T_\delta = \{(g, \delta(g)) \mid g \in G\}$ , for a map  $\delta: G \rightarrow M$ . The transversal,  $T_\delta$ , is a complement of  $\hat{M}$  in  $G \ltimes M$  if and only if  $(g, \delta(g))(h, \delta(h)) = (gh, \delta(gh))$  for all  $g, h \in G$  or, equivalently,

$$\delta(gh) = \delta(g)^h + \delta(h) \quad \forall g, h \in G. \quad (2.6)$$

If  $M$  is a  $KG$ -module, then a map  $\delta: G \rightarrow M$  is called a *derivation* or a *crossed homomorphism* or a *1-cocycle* if Equation (2.6) holds. Observe that setting  $h = 1_G$  in Equation (2.6) yields  $\delta(1_G) = 0_M$  for any derivation,  $\delta$ .

The set,  $\{\delta: G \rightarrow M\}$ , of derivations is denoted by  $Z^1(G, M)$ . By using the obvious pointwise addition and scalar multiplication,  $Z^1(G, M)$  can be made into a  $K$ -module. The set,  $T_\delta$ , is a complement of  $\hat{M}$  in  $G \ltimes M$  if and only if  $\delta \in Z^1(G, M)$ .

Notice that for a fixed  $m \in M$ ,  $\{(g, 0_M)^{(1_G, m)} = (g, m - m^g) \mid g \in G\}$  is a complement of  $\hat{M}$  in  $G \ltimes M$ , and so  $g \mapsto m - m^g$  is a derivation. Such a map is called a *principal derivation* or *1-coboundary*. The set of all principal derivations is denoted by  $B^1(G, M)$  and forms a  $K$ -submodule of  $Z^1(G, M)$ . The quotient  $K$ -module,  $H^1(G, M) = Z^1(G, M)/B^1(G, M)$ , is called the *first cohomology group* of  $G$ ,  $M$  and the associated action. By construction,  $H^1(G, M)$  is in one-one correspondence with the set of conjugacy classes of complements of  $\hat{M}$  in  $G \ltimes M$ .

### 2.3.3 Extensions of Modules and The Second Cohomology Group

Let  $E$  be any extension of an abelian subgroup,  $M$ , (regarded as a subgroup of  $E$ ) by a group,  $G$ . So there exists an epimorphism  $\rho: E \rightarrow G$  with kernel  $M$ . For  $g \in G$ , choose  $\hat{g} \in E$  with  $\rho(\hat{g}) = g$  and, for  $m \in M$ , define  $m^g = m^{\hat{g}}$ . Since  $M$  is abelian, this definition is independent of the choice of  $\hat{g}$ , and it defines an action of  $G$  on  $M$ . In general, this action makes  $M$  into a  $\mathbb{Z}G$ -module, but if  $M$  happens to be a

module over a commutative ring  $K$  with unity, and the conjugation actions of  $g \in G$  define  $K$ -automorphisms of  $M$ , then  $M$  becomes a  $KG$ -module. In particular, this is true with  $K = \mathbb{F}_p$  in the case when  $M$  is an elementary abelian  $p$ -group for some prime,  $p$ .

**Definition 2.12.** Let  $G$  be a group and  $M$  a  $KG$ -module for some commutative ring,  $K$ . A  $KG$ -module extension of  $M$  by  $G$  is defined to be a group extension,  $E$ , of  $M$  by  $G$  in which the given  $KG$ -module,  $M$ , is the same as the  $KG$ -module defined by conjugation within  $E$ .

Given  $E$  as above, the set,  $\{\hat{g} \mid g \in G\}$ , forms a transversal of  $M$  in  $G$ . For  $g, h \in G$ , one has  $\hat{g}\hat{h} = \widehat{gh}\tau(g, h)$ , for some function,  $\tau: G \times G \rightarrow M$ , where the associative law in  $E$  implies that, for all  $g, h, k \in G$ ,

$$\tau(g, hk) + \tau(h, k) = \tau(g, h)^k + \tau(gh, k).$$

A function  $\tau: G \times G \rightarrow M$  satisfying this identity is called a *2-cocycle*, and the additive group of such functions forms a  $K$ -module and is denoted by  $Z^2(G, M)$ .

Conversely, it is straightforward to check that, for any  $\tau \in Z^2(G, M)$ , the group  $E = \{(g, m) \mid g \in G, m \in M\}$  with multiplication defined by

$$(g_1, m_1)(g_2, m_2) = (g_1g_2, \tau(g_1, g_2) + m_1^{g_2} + m_2)$$

is a  $KG$ -module extension of  $M$  by  $G$  that defines the 2-cocycle  $\tau$  on choosing  $\hat{g} = (g, 0)$ .

A general transversal of  $M$  in  $E$  has the form  $\hat{g} = (g, \delta(g))$  for a function,  $\delta: G \rightarrow M$ , and it can be checked that this transversal defines the 2-cocycle,  $\tau + c_\delta$ , where  $c_\delta$  is defined by  $c_\delta(g, h) = \delta(gh) - \delta(g)^h - \delta(h)$ . A 2-cocycle of the form,  $c_\delta$ , for a function,  $\delta: G \rightarrow M$ , is called a *2-coboundary*, and the additive group of such functions is a  $K$ -module and it denoted by  $B^2(G, M)$ .

Two  $KG$ -module extensions,  $E_1$  and  $E_2$  of a  $KG$ -module,  $M$ , by  $G$  are said to be *equivalent* if there is an isomorphism from  $E_1$  to  $E_2$  that maps the copy of  $M$  in  $E_1$  to the copy of  $M$  in  $E_2$ , and induces the identity map on both  $M$  and on  $G$ . From the above discussion, it is not difficult to show that the extensions,  $E_1$  and  $E_2$  with respective 2-cocycles,  $\tau_1$  and  $\tau_2$  are equivalent if and only if  $\tau_1 - \tau_2 \in B^2(G, M)$  and, in particular, an extension splits if and only if its corresponding 2-cocycle belongs to  $B^2(G, M)$ .

The quotient  $K$ -module,  $H^2(G, M) = Z^2(G, M)/B^2(G, M)$ , is called the *second cohomology group* of  $G$ ,  $M$  and the associated action. It follows from the discussion above that  $H^2(G, M)$  is in one-one correspondence with the equivalence classes of  $KG$ -module extensions of  $M$  by  $G$ .

# Chapter 3

## Multiplication

Multiplication is the most fundamental operation that one can perform within a group. In order to design a multiplication algorithm which produces consistent results, a normal form for group elements must be defined.

This chapter contains a detailed description of the multiplication algorithm developed for the class of polycyclic-by-finite groups.

Firstly, the proposed normal form for elements of polycyclic-by-finite groups is introduced. After a brief analysis of the technicalities involved in designing a feasible multiplication method, the data structure used to represent polycyclic-by-finite groups is presented, followed by the multiplication algorithm itself. The chapter concludes with a survey of useful functions that follow as straightforward applications of the multiplication method.

The definition of the normal form, and the subsequent theory developed, relies on the presupposition that it is computationally feasible to represent the finite quotient of the polycyclic-by-finite group in question faithfully by a group of permutations or matrices. Specifically, a base and strong generating set data structure for the quotient is required. Thus, it shall hereinafter be assumed that, in all cases, such a representation exists and, for the sake of clarity, the finite quotient shall be viewed as a permutation group.

## 3.1 Representation of Elements

The preliminary aspects of computing with polycyclic-by-finite groups are discussed in this section.

### 3.1.1 The Normal Form

The first stage in representing polycyclic-by-finite groups on the computer involves defining a suitable normal form for elements of such groups. The design of such a normal form can be based on the structure of the group as an extension of a polycyclic group by a finite group, and should facilitate effective manipulation of elements. In particular, it must be possible to multiply elements written in normal form efficiently.

Let  $E$  be a polycyclic-by-finite group, let  $N \triangleleft E$  be polycyclic of finite index in  $E$  and denote the quotient  $E/N$  by  $G$ . An element  $e \in E$  can be uniquely represented as an ordered pair  $(g, n)$  where  $g \in G$  and  $n \in N$ , and conversely each such ordered pair determines an element of  $E$ . A base and strong generating set data structure for the permutation group  $G$ , combined with a power-conjugate or polycyclic presentation for  $N$ , automatically induce a normal form for elements of  $E$  written in this manner.

The ordered pair normal form is observed to be an efficacious way of storing elements of a polycyclic-by-finite group, for its components are elements of a finite permutation group and a polycyclic group respectively — groups for which there are well-developed, optimised algorithms available, thereby fully exploiting the structure of the group in question.

In the context of computing with algebraic structures, representing the polycyclic-by-finite group  $E$  as a pair  $(G, N)$  (with an associated action) not only provides one with a tractable form for elements, but also elucidates structural information about the group. For instance, a check for solubility amounts simply to testing  $G$  for the

property. The transparent nature of the representation aids in the design of potentially complex algorithms whose operation centres around structural computation.

### 3.1.2 The Multiplication Problem

Let  $E$ ,  $N$  and  $G$  be as in Subsection 3.1.1, let  $\rho: E \rightarrow G$  be the natural map, and fix a left transversal,  $L$ , of  $N$  in  $E$  with  $1_E \in L$ . For each  $g \in G$ , denote the unique element of  $\rho^{-1}(g) \cap L$  by  $\bar{g}$ .

Let  $e \in E$  and suppose that  $\rho(e) = g$ . Then  $e = \bar{g}n$ , where  $n \in N$  is uniquely determined by the left transversal,  $L$ , and, as in Subsection 3.1.1,  $e$  can be represented by the ordered pair  $(g, n)$ .

Consider multiplying two such elements,  $e_1 = \bar{g}_1 n_1$  and  $e_2 = \bar{g}_2 n_2$ :

$$e_1 \cdot e_2 = (\bar{g}_1 n_1) \cdot (\bar{g}_2 n_2) = \bar{g}_1 \bar{g}_2 \cdot n_1^{\bar{g}_2} n_2.$$

The first component in the ordered pair representation of the product  $e_1 \cdot e_2$  is  $g = \rho(e_1 \cdot e_2) = g_1 g_2$ . The second component (relative to the transversal,  $L$ ) is thus

$$(\overline{g_1 g_2})^{-1} e_1 e_2 = (\overline{g_1 g_2})^{-1} \bar{g}_1 \bar{g}_2 \cdot n_1^{\bar{g}_2} n_2 = n_{(e_1, e_2)} \in N,$$

and the product  $e_1 \cdot e_2$  is represented by the ordered pair  $(g, n_{(e_1, e_2)})$ .

The computation above illustrates the difficulty in formulating a feasible multiplication algorithm for elements of  $E$  written as ordered pairs. For any pair  $e_1, e_2$  of elements of  $E$ , such an algorithm must be able to compute the corresponding  $n_{(e_1, e_2)} \in N$  without performing any element arithmetic in  $E$ . This is achieved by precomputing and storing these values for a set of key elements, as described in the following section.



## 3.2 The Multiplication Algorithm

This section contains a detailed description of the multiplication method.

### 3.2.1 Strategy

The approach used to solve the multiplication problem is described here.

Keeping the notation of Subsection 3.1.2, let  $S = \{x_1, \dots, x_m\}$  be a non-redundant strong generating set relative to a base,  $B = (\beta_1, \dots, \beta_l)$ , for the finite group,  $G$ . For each  $i$ , denote the  $i$ -th basic stabiliser relative to  $B$  by  $G^{[i]}$  and denote the  $i$ -th basic orbit relative to  $B$  by  $\Delta_i = \{\delta_{i,1}, \dots, \delta_{i,d_i}\}$ , where  $\delta_{i,1} = \beta_i$ . Additionally, let  $S_i = S \cap G^{[i]} = \{x_{i_1}, \dots, x_{i_{s_i}}\}$  and denote by  $S_i^{-1}$  the set,  $\{x_{i_1}^{-1}, \dots, x_{i_{s_i}}^{-1}\}$ .

For each  $i$ , let  $U_i$  be a right transversal of  $G^{[i+1]}$  in  $G^{[i]}$ . Then, as described in Subsection 2.1.3, each element of  $G$  can be represented uniquely as a product of transversal elements  $u_l u_{l-1} \cdots u_1$  where  $u_i \in U_i$ .

Fix  $i$  and let  $u \in U_i$  be the permutation taking  $\beta_i$  to  $\delta_{i,j} \in \Delta_i$ . Take  $x \in S_i$ , and let  $h, h'$  be the permutations in  $U_i$  which map  $\beta_i$  to  $\beta_i^{ux} = \delta_{i,j}^x$ ,  $\beta_i^{ux^{-1}} = \delta_{i,j}^{x^{-1}}$  respectively. Then

$$u \cdot x = y_1 y_2 \cdots y_k \cdot h, \quad (3.1a)$$

$$u \cdot x^{-1} = z_1 z_2 \cdots z_{k'} \cdot h', \quad (3.1b)$$

where  $y_1, \dots, y_k, z_1, \dots, z_{k'}$  are elements of  $S_{i+1} \cup S_{i+1}^{-1}$ . The words,  $y_1 y_2 \cdots y_k$  and  $z_1 z_2 \cdots z_{k'}$ , are called the *tails* of  $u$  relative to  $x$  and  $x^{-1}$  respectively.

Now regard  $G$  as a quotient of the larger group,  $E$ . Let  $L$  be a left transversal of  $N$  in  $E$  with  $1_E \in L$  and, as in Subsection 3.1.2, for each  $g \in G$ , denote the unique

element of  $g \cap L$  by  $\bar{g}$ . Then Equations (3.1) become

$$\bar{u} \cdot \bar{x} = \bar{y}_1 \bar{y}_2 \cdots \bar{y}_k \cdot \bar{h} \cdot n, \quad (3.2a)$$

$$\bar{u} \cdot \overline{x^{-1}} = \bar{z}_1 \bar{z}_2 \cdots \bar{z}_{k'} \cdot \bar{h}' \cdot n', \quad (3.2b)$$

respectively, for some  $n, n' \in N$ . The elements  $n$  and  $n'$  are called the *heads* of  $u$  relative to  $x$  and  $x^{-1}$  respectively and the Equations (3.2) are called the *shift* equations.

The shift equations suggest a scheme by which elements of  $E$  may be multiplied. Suppose that the elements,  $\bar{u}_i$  (where  $u_i \in U_i$  for each  $i$ ), and the tails,  $y_1 y_2 \cdots y_k$ ,  $z_1 z_2 \cdots z_{k'}$ , can be computed consistently. Furthermore, suppose that the conjugate of each element of the normal subgroup,  $N$ , by elements,  $\bar{x}$  and  $\overline{x^{-1}}$  (where  $x \in S$ ), can be calculated without performing any element arithmetic in  $E$ . Then the shift equations can be utilised to design an iterative function which performs multiplications of the form:

$$(\bar{u}_l \overline{u_{l-1}} \cdots \bar{u}_1 n_1) \cdot (\bar{x} n_2)$$

where  $n_1, n_2 \in N$ , conjugating within  $N$  whenever necessary. This method can be extended to handle element multiplications in full generality.

The tasks of conjugation in  $N$  and calculation of the elements,  $\bar{u}_i \in L$ , are performed via a precomputed data set, the specification of which forms the content of the next subsection.

### 3.2.2 Data

The multiplication algorithm relies directly on a precomputed set of data to calculate the elements of the normal polycyclic subgroup that arise when elements of the chosen transversal are shifted past strong generator preimages in the polycyclic-by-

finite group. A technical description of the contents of this data set is given in this subsection.

### The Finite Quotient

The first segment of required data is computed wholly within the finite quotient  $G$  of the given polycyclic-by-finite group. For each  $i \in \{1, \dots, |B|\}$  and each  $\delta_{i,j} \in \Delta_i$ , the tails of  $u$ , the permutation in  $U_i$  mapping  $\beta_i$  to  $\delta_{i,j}$ , relative to each  $x \in S_i$  and  $x^{-1} \in S_i^{-1}$  are calculated and stored.

In the implementation, a fixed Schreier vector data structure is used to encode the transversal elements,  $u_i \in U_i$ , as words over  $S \cup S^{-1}$ . Since this data structure remains unchanged, a given permutation in  $G$  can be written as a word over  $S \cup S^{-1}$  consistently by using the set of stored Schreier vectors to expressing said permutation in normal form  $u_l u_{l-1} \cdots u_1$  (where  $u_i \in U_i$ ), storing each  $u_i$  as an integer sequence representing a word over  $S \cup S^{-1}$ .

Tails are computed for each  $i \in \{1, \dots, |B|\}$  and each element in  $\Delta_i$ , for each applicable strong generator. It follows from the inequalities,

$$|B| \leq \log_2 |G|,$$

$$|\Delta_i| \leq n$$

and

$$|S_i| \leq |S|,$$

that the number of tails is  $O(|S|n \log_2 |G|)$ . The calculation of each tail requires one application of the sifting procedure, which has time complexity  $O(n \log_2 |G|)$  (see Subsection 2.1.3). Hence, the time complexity of computing all tails is  $O(|S|(n \log_2 |G|)^2)$ .

A tail is formed by concatenating  $|B|$  integer sequences, each of length less than  $n$ , and so, each tail requires  $O(n \log_2 |G|)$  storage, hence the space complexity

associated with storing all tail elements is identical to its time complexity, namely  $O(|S|(n \log_2 |G|)^2)$ .

All algorithms presented in this chapter, and indeed in the rest of the thesis, assume the existence of a method which operates as outlined to write permutations of  $G$  as words over  $S \cup S^{-1}$ , and the tails of each  $u$ , relative to each  $x$ ,  $x^{-1}$  are computed in a consistent manner using this method. This uniformity is critical when passing from the quotient,  $G$ , to the polycyclic-by-finite group,  $E$ .

### The Polycyclic-by-Finite Group

Three distinct sectors of data are computed by performing element arithmetic within the polycyclic-by-finite group in question, using its existing representation.

Firstly, the heads of the shift equations are computed and recorded. In order to do so consistently, an explicit definition of the transversal,  $L$ , is required. The construction is as follows. For each  $x \in S$ , the element  $\bar{x} \in L$  is fixed by choosing exactly one member of each corresponding preimage set  $\rho^{-1}(x)$ . The set

$$\bar{S} = \{\bar{x} \mid x \in S\}$$

is held in memory and is used to define the inverse image set

$$\overline{S^{-1}} = \{\overline{x^{-1}} \mid x \in S\}$$

by the equation:

$$\overline{x^{-1}} = \begin{cases} \bar{x} & \text{if } x^2 = 1, \\ \bar{x}^{-1} & \text{otherwise.} \end{cases}$$

Identifying strong generators of order 2 as a special case is necessary to eliminate the possibility of a strong generator being assigned two distinct images in  $L$  which would violate uniqueness of coset representatives in a transversal.

The remaining elements of  $L$  are calculated when required using the sets  $\overline{S}$  and  $\overline{S}^{-1}$ , as prescribed by the Schreier vector data structure held in memory. Given a permutation  $g \in G$ , the transversal element  $\overline{g} \in L$  is computed by first expressing  $g$  as a word over  $S \cup S^{-1}$  (via the Schreier vector data structure), and then multiplying through this word, substituting recorded values from the sets  $\overline{S}$  and  $\overline{S}^{-1}$ .

As discussed above, the Schreier vector data structure furnishes one with a method by which a permutation in  $G$  can be consistently represented as word over  $S \cup S^{-1}$ . This ensures that the method proposed here for computing transversal elements, in turn, produces consistent results.

For each  $u, x$  pair as above,  $n$  and  $n'$  of Equations 3.2 are then calculated in the obvious manner:

$$n = \overline{h}^{-1} \cdot \overline{y_k}^{-1} \overline{y_{k-1}}^{-1} \cdots \overline{y_1}^{-1} \cdot \overline{u} \cdot \overline{x}, \quad (3.3a)$$

$$n' = \overline{h'}^{-1} \cdot \overline{z_{k'}}^{-1} \overline{z_{k'-1}}^{-1} \cdots \overline{z_1}^{-1} \cdot \overline{u} \cdot \overline{x}^{-1}. \quad (3.3b)$$

The products above are evaluated using the existing representation for  $E$ .

Each tail is of length  $O(n \log_2 |G|)$ . Thus, the calculation of the heads,  $n$  and  $n'$ , above require  $O(n \log_2 |G|)$  multiplications using the existing representation of  $E$ ; for all of the  $O(|S|n \log_2 |G|)$  tails, the total time cost is therefore  $O(|S|(n \log_2 |G|)^2)$ .

This step of data collection stores one element of the polycyclic group,  $N$ , for each of the  $O(|S|n \log_2 |G|)$  tails. Assuming that an element of  $N$  requires  $O(r)$  storage, the memory cost here is  $O(|S|rn \log_2 |G|)$ .

The second portion of data computed within the polycyclic-by-finite group relates to conjugation of elements of the normal subgroup  $N$  by elements of  $E$ . During its operation, the multiplication algorithm frequently calls for conjugations of the form  $n^w$ , where  $n \in N$  and  $w$  is a word over  $\overline{S} \cup \overline{S}^{-1}$ . Thus, it is necessary that such elements be computable without appealing to the original representation of the

group  $E$ . To enable such functionality, the conjugates

$$a^{\bar{x}}, a^{\overline{x^{-1}}} \in N$$

are computed and stored for each polycyclic generator,  $a$ , of the stored polycyclic presentation for  $N$ , and each  $x \in S$ . This information is sufficient to devise a straightforward iterative procedure which calculates conjugates,  $n^w$ , without performing element arithmetic in  $E$ . During this step,  $O(r|S|)$  multiplications are performed using the existing representation of  $E$ , where  $r$  is the size of the stored polycyclic generating sequence for  $N$ . Assuming  $O(r)$  storage for each of the  $O(r|S|)$  elements of  $N$ , the total memory requirement is  $O(r^2|S|)$ .

Finally, sets,  $\hat{S}$  and  $\hat{S}^{-1}$ , representing the elements of  $\bar{S}$  and  $\overline{S^{-1}}$  respectively in normal form are stored. In this case, these are:

$$\hat{S} = \{(y, 1_N) : y \in S\}, \tag{3.4a}$$

$$\hat{S}^{-1} = \{(y, 1_N) : y \in S^{-1}\}. \tag{3.4b}$$

Computing an element of  $\hat{S}$  or  $\hat{S}^{-1}$  requires one application of the sifting procedure, which has time complexity  $O(n \log_2 |G|)$ , so the time complexity associated with this step of data collection is  $O(|S|n \log_2 |G|)$ .

The first component of the normal form for an element of  $E$  consists of  $|B|$  integer sequences, each of length less than  $n$ , and thus requires  $O(n \log_2 |G|)$  storage. The second component of the normal form is an element of  $N$ , for which it is assumed that  $O(r)$  storage is necessary. Thus,  $\hat{S}$  and  $\hat{S}^{-1}$  require  $O(|S|(r + n \log_2 |G|))$  storage.

### 3.2.3 The Algorithm

Equipped with the data set of Subsection 3.2.2, it is possible to design an algorithm which, when given two elements of  $E$  written in the normal form defined in Sub-

section 3.1.1, computes the normal form of their product without performing any extemporaneous element arithmetic in  $E$ .

Let  $e_1$  and  $e_2$  be two elements of  $E$  with normal form representations,  $(g_1, n_1)$  and  $(g_2, n_2)$ , respectively. Keeping the notation of Subsection 3.2.1, write  $e_1$  as

$$(\overline{u_l} \overline{u_{l-1}} \cdots \overline{u_1}) \cdot n_1$$

where  $u_i \in U_i$  for each  $i$ , and  $n_1 \in N$ . Let  $e_2 = \overline{g_2} n_2$  where  $n_2 \in N$ ,  $\overline{g_2} \in L$ .

The multiplication algorithm initially expresses  $\overline{g_2}$  as a word over  $\overline{S} \cup \overline{S^{-1}}$ , say  $w = \overline{q_1} \overline{q_2} \cdots \overline{q_\sigma}$  where  $q_j \in S \cup S^{-1}$  for each  $j$ . The desired product may then be written as

$$e_1 \cdot e_2 = (\overline{u_l} \overline{u_{l-1}} \cdots \overline{u_1}) \cdot n_1 \cdot (\overline{q_1} \overline{q_2} \cdots \overline{q_\sigma}) \cdot n_2$$

The algorithm proceeds to rearrange the terms of the product by conjugating the element  $n_1$  by the word  $w$ . As discussed in Subsection 3.2.2, this conjugation is performed using a precomputed data set, with no element arithmetic carried out in  $E$ . The product then becomes

$$e_1 \cdot e_2 = (\underbrace{\overline{u_l} \overline{u_{l-1}} \cdots \overline{u_1}}_{\text{under-bracketed}}) \cdot \overline{q_1} \overline{q_2} \cdots \overline{q_\sigma} \cdot n_1^{\overline{q_1} \overline{q_2} \cdots \overline{q_\sigma}} n_2 \quad (3.5)$$

$$= \overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2} \underbrace{\overline{u_1} \cdot \overline{q_1}}_{\text{under-bracketed}} \overline{q_2} \cdots \overline{q_\sigma} \cdot n_1^w n_2. \quad (3.6)$$

**Note 3.1.** Initially, the leftmost sequence of transversal elements,  $\overline{u_i}$ , in the expression for the product (under-bracketed in Equation (3.5)) has  $l$  terms.

The under-bracketed segment of Equation (3.6) can be recognised as the left-hand side of a shift equation, say

$$\overline{u_1} \cdot \overline{q_1} = \overline{y_1} \overline{y_2} \cdots \overline{y_\kappa} \cdot \overline{h_1} \cdot n_3 \quad (3.7)$$

for some  $n \in N$ ,  $h_1 \in U_1$  and where  $y_j \in S_2 \cup S_2^{-1}$  for each  $j$ . Via direct substitution, the Equation (3.7) can be used to “shift” the transversal element,  $\overline{q_1}$ , past  $\overline{u_1}$  without performing any element arithmetic in the group  $E$ . The algorithm executes this shifting procedure as illustrated below.

$$\begin{aligned}
 e_1 \cdot e_2 &= \overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2} \underbrace{\overline{u_1} \cdot \overline{q_1}}_{\text{replace by } \overline{y_1} \overline{y_2} \cdots \overline{y_\kappa} \cdot \overline{h_1} \cdot n_3} \overline{q_2} \cdots \overline{q_\sigma} \cdot n_1^w n_2 \\
 &= \overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2} \cdot (\overline{y_1} \overline{y_2} \cdots \overline{y_\kappa} \cdot \overline{h_1} \cdot n_3) \cdot \overline{q_2} \overline{q_3} \cdots \overline{q_\sigma} \cdot n_1^w n_2 \\
 &= \overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2} \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_\kappa} \cdot \underbrace{\overline{h_1} \cdot \overline{q_2}} \overline{q_3} \cdots \overline{q_\sigma} \cdot n_3^{\overline{q_1}^{-1}w} n_1^w n_2
 \end{aligned} \tag{3.8}$$

The final rearrangement in Equation (3.8) is the result of conjugating  $n_3$  by

$$\overline{q_2} \overline{q_3} \cdots \overline{q_\sigma} = \overline{q_1}^{-1} w.$$

The under-bracketed segment of the product in Equation (3.8) can again be recognised as the left-hand side of a shift equation. The algorithm repeats the procedure above to shift  $\overline{q_2}$  past  $\overline{h_1}$ .

**Observation 3.2.** Each time a shift is made, the word over  $\overline{S} \cup \overline{S}^{-1}$  immediately to the left of the sequence of normal subgroup elements in the expression for the product is reduced by exactly one symbol.

In light of Observation 3.2, the algorithm is able to continue iterating through the word,  $w$ , shifting at every step, arriving at the following equation:

$$\begin{aligned}
 e_1 \cdot e_2 &= \overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2} \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_\zeta} \cdot \overline{u'_1} \cdot n_{\sigma+2} n_{\sigma+1}^{\overline{q_\sigma}} \cdots n_3^{\overline{q_1}^{-1}w} n_1^w n_2 \\
 &= \underbrace{\overline{u_l} \overline{u_{l-1}} \cdots \overline{u_2}} \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_\zeta} \cdot \overline{u'_1} \cdot n',
 \end{aligned} \tag{3.9}$$

for some  $u'_1 \in U_1$ ,  $n_3, \dots, n_{\sigma+2} \in N$ ; where  $y_j \in S_2 \cup S_2^{-1}$  for each  $j$  and

$$n' = n_{\sigma+2} n_{\sigma+1}^{\overline{q_\sigma}} \cdots n_3^{\overline{q_1}^{-1}w} n_1^w n_2.$$



At this stage, the algorithm has decreased the under-bracketed sequence of transversal elements,  $\overline{u_i}$ , in Equation (3.5) by one term, as indicated in Equation (3.9), thus reducing the problem to a smaller case. The algorithm restarts its inner loop to process the word,  $\overline{y_1 y_2 \cdots y_\zeta}$ , by, as before, recognising  $\overline{u_2} \cdot \overline{y_1}$  as the left-hand side of a shift equation.

**Observation 3.3.** Each time a word over  $\overline{S} \cup \overline{S^{-1}}$  is processed fully as described above, the leftmost sequence of transversal elements,  $\overline{u_i}$ , in the expression for the product is decreased by exactly one term.

The word-processing procedure is repeated for each  $\overline{u_i}$  yielding an expression of the form

$$\overline{u'_l} \overline{u'_{l-1}} \cdots \overline{u'_1} \cdot n''$$

where  $u'_i \in U_i$  for each  $i$ , for the product.

**Remark 3.4.** Whenever a shift is made at level  $i$  (with respect to the base and strong generating set hierarchy), the elements of  $\overline{S} \cup \overline{S^{-1}}$  that are placed to the left of the new element,  $\overline{h_i}$  (where  $h_i \in U_i$ ), belong to  $\overline{S_{i+1}} \cup \overline{S_{i+1}^{-1}}$ . In particular, since  $S_{l+1} = \emptyset$ , no non-trivial elements of  $\overline{S} \cup \overline{S^{-1}}$  are placed to the left of any possible  $\overline{h_l}$  where  $h_l \in U_l$ .

The multiplication method is presented in Algorithm 3.1. The function takes as input two elements,  $(g_1, n_1)$ ,  $(g_2, n_2)$ , written in normal form, of a polycyclic-by-finite group,  $E$ , and operates as described above to compute the normal form for the product  $(g_1, n_1) \cdot (g_2, n_2)$ .

The notation employed thus far shall be used in the description of the algorithm, viz. the normal polycyclic subgroup of  $E$  and associated quotient, via which the normal form is defined, shall be denoted by  $N$  and  $G$  respectively. It is assumed during execution that the data set of Subsection 3.2.2 is held in memory, with the tails of the shift equations stored as arrays of strong generators. The base

and strong generating set relative to which the data is computed are denoted by  $S = \{x_1, \dots, x_m\}$  and  $B = (\beta_1, \dots, \beta_l)$  respectively, and the left transversal of  $N$  in  $E$  by  $L$ .

The algorithm also assumes the existence of a function `LENGTH` which, when supplied with an array (or sequence), returns the number of non-null entries of that array (or sequence).

The central step in measuring the run-time of `MULTIPLY` is counting the number of times that Lines 16–21 are executed. To calculate this, one must establish a bound for `LENGTH(rightword)`, for each of the  $l$  iterations of the outer loop. Such a bound is derived in the proof of Theorem 3.5. The most expensive computations within Lines 16–21 occur in Line 18 and in Line 20. Line 18 consists of an application of a straightforward iterative procedure to compute the conjugate of an element of  $N$  by an element of  $\overline{S} \cup \overline{S}^{-1}$  using the precomputed data set (see Subsection 3.2.2). Line 20 is a standard Schreier vector computation in  $G$ , whose time complexity is discussed in Subsection 2.1.2. The operations of Line 18 and Line 20 dominate the memory retrievals and variable reassignment in Lines 16–21, and so, in Theorem 3.5, the time complexity of `MULTIPLY` is stated in terms of the number of multiplications that must be performed within the finite quotient, and conjugations of elements of the normal polycyclic subgroup by preimages of strong generators (and inverses).

**Theorem 3.5.** *The multiplication algorithm terminates with the correct value for the desired product, and requires  $O(n(n \log_2 |G|)^{\log_2 |G|})$  multiplications using the permutation representation of the finite quotient,  $G$ , and  $O((n \log_2 |G|)^{\log_2 |G|})$  conjugations of elements of the normal polycyclic subgroup by strong generator preimages, where  $n$  is the degree of the permutation representation of  $G$ .*

*Proof.* Termination is guaranteed by Observations 3.2 and 3.3 combined with Remark 3.4, while Equations (3.5)–(3.9) imply correctness.

The rate of growth of *rightword* is investigated as follows. Subsection 3.2.2 shows

**Algorithm 3.1** Element Multiplication

---

```

1: function MULTIPLY( $(g_1, n_1), (g_2, n_2)$ )
2:   if  $g_2 = 1_G$  then
3:     return  $(g_1, n_1 n_2)$ 
4:   end if
5:   Write  $g_1$  in normal form  $u_l u_{l-1} \cdots u_1$  where  $u_i \in U_i$  for each  $i$ 
6:   Write  $g_2$  as a word,  $w = q_1 q_2 \cdots q_t$ , where  $q_i \in S \cup S^{-1}$  for each  $i$ 
7:    $rightword \leftarrow [q_1, q_2, \dots, q_t]$ 
8:    $leftword \leftarrow []$ 
9:    $n'_1 \leftarrow n_1$ 
10:  for  $i \leftarrow 2$  to  $l$  do
11:     $n'_i \leftarrow 1_N$ 
12:  end for
13:  for  $i \leftarrow 1$  to  $l$  do
14:     $u'_i \leftarrow u_i$ 
15:    for  $j \leftarrow 1$  to LENGTH( $rightword$ ) do
16:      Retrieve from memory the tail array,  $tailword$ , and tail element,  $n_\sigma$ ,
of the shift equation corresponding to the pair:  $u'_i, rightword[j]$ 
17:       $leftword \leftarrow leftword \text{ cat } tailword$ 
18:      Find the conjugate,  $n_\gamma$ , of  $n'_i$  by the image of  $rightword[j]$  in  $L$ 
19:       $n'_i \leftarrow n_\sigma \cdot n_\gamma$ 
20:      Find  $h \in U_i$  which maps  $\beta_i^{u'_i}$  to the image of  $\beta_i^{u'_i}$  under  $rightword[j]$ 
21:       $u'_i \leftarrow h$ 
22:    end for
23:     $rightword \leftarrow leftword$ 
24:     $leftword \leftarrow []$ 
25:  end for
26:   $g_* \leftarrow 1_G, n_* \leftarrow 1_N, i \leftarrow l$ 
27:  while  $i > 0$  do
28:     $n_* \leftarrow \overline{n_*^{u'_i}} \cdot n'_i$   $\triangleright$  The conjugation  $\overline{n_*^{u'_i}}$  is performed using the data set
29:     $g_* \leftarrow g_* \cdot u'_i$ 
30:     $i \leftarrow i - 1$ 
31:  end while
32:   $n_* \leftarrow n_* \cdot n_2$ 
33:  return  $(g_*, n_*)$ 
34: end function

```

---

that, in the first iteration of the outer loop of MULTIPLY, *rightword* has length at most  $n \log_2 |G|$ . As illustrated in Subsection 3.2.3, each shift made in the inner loop of the algorithm appends a tail of length at most  $n \log_2 |G|$  to the variable, *leftword*. This operation is performed in Line 17. Since a shift is made for each element of *rightword*, it follows that, at the end of the first iteration of the outer loop, *leftword* has length at most

$$\text{LENGTH}(\textit{rightword}) \cdot n \log_2 |G| \leq (n \log_2 |G|)^2.$$

Hence, in the second iteration of the outer loop, *rightword* has length at most  $(n \log_2 |G|)^2$ . Applying an similar argument, one may deduce that, in the third iteration of the outer loop *rightword* has length at most  $(n \log_2 |G|)^3$ , and, in general, *rightword* has length at most  $(n \log_2 |G|)^i$  in the  $i$ -th iteration of the outer loop.

Thus, in a run of MULTIPLY, Lines 16–21 are executed no more than

$$\sum_{i=1}^l (n \log_2 |G|)^i$$

times, where  $l$  is the length of the base of the finite quotient,  $G$ .

By virtue of the inequality  $l \leq \log_2 |G|$ , the sum,  $\sum_{i=1}^l (n \log_2 |G|)^i$  is  $O((n \log_2 |G|)^{\log_2 |G|})$ .

It follows immediately that MULTIPLY performs  $O((n \log_2 |G|)^{\log_2 |G|})$  conjugations of polycyclic group elements by strong generator preimages.

As discussed in Subsection 2.1.2, the Schreier vector computation of Line 20 requires  $O(n)$  multiplications in the finite quotient,  $G$ . Thus a run of MULTIPLY requires a total of  $O(n(n \log_2 |G|)^{\log_2 |G|})$  multiplications in  $G$ .  $\square$

In light of the exponential growth (illustrated in the proof Theorem 3.5) of the words through which MULTIPLY must iterate, small base representations for the quotient group are highly desirable for effective performance of the multiplication

algorithm. As discussed in Subsection 1.2.3 of Chapter 1, in constructing the representation for the given polycyclic-by-finite group, one attempts to “factor out” the largest possible  $N$ , minimising  $|G|$  and consequently restricting the size of any base  $B$  for  $G$  via the inequality  $|B| \leq \log_2 |G|$  (see Subsection 2.1.3). Finding a quotient which admits a base of moderate size is a critical safeguard against the potential run-time cost associated with the accumulation of long words.

In practice, sparing cases where an injudicious choice is made for  $N$  (and hence  $G$ ), the arrays that appear are of manageable span, yielding positive empirical results.

Multiplication of elements written in normal form shall hereinafter be indicated with the use of standard infix notation, omitting the ‘.’ operator when context permits.

### 3.3 Applications

With element arithmetic in place, one may design straightforward functions to perform standard operations on an element of  $E$  written in normal form, such as finding its image in the transversal,  $L$ , computing its inverse and calculating its order. Methods to perform these tasks are discussed in Subsections 3.3.1–3.3.3.

In many applications involving subgroups of polycyclic-by-finite groups, one often encounters subgroups that are polycyclic. In such situations, a polycyclic presentation for the subgroup in question is desirable, for this would allow one to utilise the host of well-developed algorithms available for polycyclic groups. Subsection 3.3.4 describes a method which finds a consistent polycyclic (or power-conjugate) presentation for a polycyclic-by-finite group that is in fact polycyclic.

### 3.3.1 Transversal Images

Given an element,  $g \in G$ , the normal form of the element,  $\bar{g} \in L$ , is computed according to the construction of  $L$  given in Subsection 3.2.2, using the sets  $\hat{S}$  and  $\hat{S}^{-1}$  defined in Equations (3.4).

To find  $\bar{g}$ , one first expresses  $g$  as a word,  $y_1 \cdots y_k$ , over  $S \cup S^{-1}$ , by the procedure described in Subsection 3.2.2. Then the element,  $\hat{y}_i \in \hat{S} \cup \hat{S}^{-1}$ , corresponding to  $y_i$  is retrieved from memory. The normal form for  $\bar{g}$  is then given by the product:

$$\hat{y}_1 \cdots \hat{y}_k.$$

Writing the elements of  $L$  in their normal form is useful from a technical perspective, as it provides one with a method of inverting the natural map,  $\rho$ .

As in Subsection 3.2.2 and the proof of Theorem 3.5, the integer,  $k$ , is bounded above by  $n \log_2 |G|$ , so, finding a transversal image requires  $O(n \log_2 |G|)$  memory retrievals and  $O(n \log_2 |G|)$  applications of MULTIPLY.

### 3.3.2 Element Inversion

Let  $(g, n_1)$  be an element of  $E$ , written in normal form. To compute  $(g, n_1)^{-1}$ , first use the procedure outlined in Subsection 3.3.1 to write  $\overline{g^{-1}}$  in normal form, say  $(g^{-1}, n_2)$ . Then

$$(g, n_1) \cdot (g^{-1}, n_2) = (1_G, n_3),$$

and hence

$$\begin{aligned} (g, n_1)^{-1} &= (g^{-1}, n_2) \cdot (1_G, n_3)^{-1} \\ &= (g^{-1}, n_2) \cdot (1_G, n_3^{-1}) \\ &= (g^{-1}, n_2 n_3^{-1}). \end{aligned}$$

The inversion procedure performs exactly one more multiplication after writing

$\overline{g^{-1}}$  in normal form. Thus, the time complexity associated with inverting an element is identical to that of the method described in Subsection 3.3.1, namely  $O(n \log_2 |G|)$  multiplications.

### 3.3.3 Orders of Elements

Let  $(g, n)$  be an element of  $E$ , written in normal form. To calculate  $o(g, n)$ , first compute the power

$$(g, n)^{o(g)} = (1_G, n').$$

Then

$$o(g, n) = o(g)o(n').$$

Using the method of repeated squaring to compute powers, one easily observes that finding the order of an element requires  $O(\log_2 |G|)$  applications of MULTIPLY.

### 3.3.4 Transfer to the Category of Polycyclic Groups

Suppose that the polycyclic-by-finite group,  $E$ , is in fact polycyclic. Then  $G$  is polycyclic and one may use the already available methods for permutation groups to compute a power-conjugate presentation for  $G$ . A consistent polycyclic or power-conjugate presentation for  $E$  may be found by “glueing together” the presentations for  $G$  and  $N$  as described below.

For the sake of clarity, it shall be assumed in the following discussion that  $N$  (and hence  $E$ ) is finite, and a power-conjugate presentation for  $E$  shall be constructed; a similar procedure may be employed in the infinite case.

Let  $N$  and  $G$  have consistent power-conjugate presentations,

$$\begin{aligned} \langle a_1, \dots, a_r \mid a_j^{p_j} = w_{j,j} \quad \text{for } 1 \leq j \leq r, \\ a_j^{a_i} = w_{i,j} \quad \text{for } 1 \leq i < j \leq r \rangle \end{aligned}$$

and

$$\begin{aligned} \langle b_1, \dots, b_s \mid b_j^{q_j} = v_{j,j} \quad \text{for } 1 \leq j \leq s, \\ b_j^{b_i} = v_{i,j} \quad \text{for } 1 \leq i < j \leq s \rangle \end{aligned}$$

respectively, where

- (i)  $p_j, q_j$  are the least primes such that  $a_j^{p_j} \in \langle a_{j+1}, \dots, a_r \rangle$  for  $j < r$  and  $b_j^{q_j} \in \langle b_{j+1}, \dots, b_s \rangle$  for  $j < s$ , and  $a_r^{p_r}, b_s^{q_s}$  are the identity elements in  $N, G$  respectively, and
- (ii)  $w_{i,j}, v_{i,j}$  are words in the generator sets  $\{a_{i+1}, \dots, a_r\}, \{b_{i+1}, \dots, b_s\}$  respectively.

A power-conjugate presentation for  $E$  can be constructed on the set of generators  $\{b_1, \dots, b_s, a_1, \dots, a_r\}$  as follows. Notation is abused here in regarding the  $b_i$  both as elements of  $E$  and  $G$ , however, the meaning shall be clear from the context as explained below.

For each  $b_i$  in the power-conjugate presentation for  $G$ , the method of Subsection 3.3.1 is used to find the normal form of  $\overline{b_i}$ , say  $(b_i, n)$ , after which, the multiplication algorithm is used to compute the power,  $(b_i, n)^{q_i} = (b_i^{q_i}, n') = (v_{i,i}, n')$ . The power-conjugate presentation for  $N$  may be used to express  $n'$  as a word,  $a_1^{\alpha_1} \cdots a_r^{\alpha_r}$ , and thus one may (with abuse of notation) write the power relation:

$$b_i^{q_i} = v_{i,i} a_1^{\alpha_1} \cdots a_r^{\alpha_r}.$$

Power relations with left-hand-side  $a_i^{p_i}$  remain unchanged in the new presentation.

Conjugate relations are derived in a similar manner; the normal form of each element in  $\{\overline{b_1}, \dots, \overline{b_s}, \overline{a_1}, \dots, \overline{a_r}\}$  is found (the normal form of  $\overline{a_i}$  is simply  $(1_G, a_i)$ ), and the multiplication algorithm is used to perform the required conjugations according to the hierarchy induced by  $G$  and  $N$ . The resulting elements are expressed



as words over  $\{b_1, \dots, b_s, a_1, \dots, a_r\}$  as above.

**Proposition 3.6.** *The presentation so obtained is a consistent power-conjugate presentation for  $E$ .*

*Proof.* The presentation constructed clearly satisfies the conditions outlined in the definition of a power-conjugate presentation given in Subsection 2.2.1. That it is consistent follows immediately from the fact that the presentations for  $N$  and  $G$  above are consistent. It remains now to show that the presentation constructed indeed defines a group isomorphic to  $E$ .

Let  $E^*$  be the group defined by the presentation constructed over the generating set  $\{b_1, \dots, b_s, a_1, \dots, a_r\}$ .

Define the map,  $\psi: \{b_1, \dots, b_s, a_1, \dots, a_r\} \rightarrow \{\overline{b_1}, \dots, \overline{b_s}, \overline{a_1}, \dots, \overline{a_r}\}$ , by:

$$\psi(b_i) = \overline{b_i}, \quad (3.10)$$

$$\psi(a_j) = \overline{a_j}, \quad (3.11)$$

for  $i = 1, \dots, s$ ,  $j = 1, \dots, r$ . By construction, every relation in the presentation for  $E^*$  over  $\{b_1, \dots, b_s, a_1, \dots, a_r\}$  also holds in  $E$ , with the symbol,  $b_i$ , replaced by  $\psi(b_i)$  and the symbol,  $a_j$ , replaced by  $\psi(a_j)$ . Since  $\{\overline{b_1}, \dots, \overline{b_s}, \overline{a_1}, \dots, \overline{a_r}\}$  generates  $E$ ,  $\psi$  may be extended in the obvious manner to a homomorphism from  $E^*$  onto  $E$ . That is,  $E$  is a homomorphic image of  $E^*$ . It shall now be shown that  $\psi$  is in fact an isomorphism.

Suppose that

$$\psi(b_1^{\beta_1} \dots b_s^{\beta_s} a_1^{\alpha_1} \dots a_r^{\alpha_r}) = 1,$$

for some  $\beta_1, \dots, \beta_s, \alpha_1, \dots, \alpha_r$ , with  $0 \leq \beta_i < q_i$  for each  $i$ , and  $0 \leq \alpha_j < p_j$  for each  $j$ . Then,

$$\overline{b_1}^{\beta_1} \dots \overline{b_s}^{\beta_s} \cdot \overline{a_1}^{\alpha_1} \dots \overline{a_r}^{\alpha_r} = 1. \quad (3.12)$$

Equation (3.12) implies that  $\overline{b_1}^{\beta_1} \dots \overline{b_s}^{\beta_s} N$  is the identity in the quotient,  $G = E/N$ .

By construction, the element  $\overline{b_i}N$  corresponds to the element  $b_i$  in the consistent power-conjugate presentation for  $G$ . By uniqueness of words written in normal form,

$$\beta_1 = \beta_2 = \cdots = \beta_s = 0.$$

This reduces Equation (3.12) to

$$\overline{a_1}^{\alpha_1} \cdots \overline{a_r}^{\alpha_r} = 1.$$

Again, by construction,  $\overline{a_i}$  corresponds to element  $a_i$  in the consistent power-conjugate presentation for  $N$ . So by uniqueness,

$$\alpha_1 = \alpha_2 = \cdots = \alpha_r = 0.$$

Thus,  $\psi$  is injective, and  $E \cong E^*$  as required. □

Data facilitating computation of the isomorphism for transfer of elements between the two representations consists of a list containing the ordered pair normal form of each element of the set,  $\{\overline{b_1}, \dots, \overline{b_s}, \overline{a_1}, \dots, \overline{a_r}\}$ , together with a list of words over  $\{b_1, \dots, b_s, a_1, \dots, a_r\}$  defining the images of the elements,  $\overline{y}$ , where  $y \in S \cup S^{-1}$ .

# Chapter 4

## Subgroups

Virtually every structural computation performed within a group involves the manipulation of subgroups. Thus, in order to obtain useful results pertaining to a given polycyclic-by-finite group, one requires an efficient method by which subgroups may be computed.

This chapter contains a detailed description of the subgroup generation method developed for the class of polycyclic-by-finite groups.

Firstly, the complications involved in computing and representing subgroups of polycyclic-by-finite groups are discussed. The strategy used to overcome these difficulties is presented, after which, the procedure by which subgroups are constructed is described in detail. The chapter then proceeds to address the important question of membership testing in subgroups. Finally, a selection of straightforward applications of the subgroup generation method is given.

The theory developed in this chapter, and indeed the rest of the thesis, requires that the polycyclic-by-finite group from which subgroups are created is represented as an extension of a normal polycyclic subgroup by a finite group as described in Chapter 3, and that the multiplication algorithm is applicable. Thus, in what follows, it shall be assumed that all parent polycyclic-by-finite groups are so represented.

## 4.1 The Subgroup Generation Problem

The first step in defining the parameters of the subgroup generation problem involves choosing a mode of representation for subgroups. There is a logical option in the case of polycyclic-by-finite groups, for this class of groups is closed under formation of subgroups, and hence it is possible, at least in theory, to represent generic subgroups using the scheme developed in Chapter 3.

With this choice for the representation, generating a subgroup entails computing a data set which contains sufficient information to apply the multiplication technique of Chapter 3 wholly within the context of that subgroup. The problem may be formulated more concretely as follows:

Let  $E$  be a polycyclic-by-finite group, let  $N \triangleleft E$  be polycyclic of finite index in  $E$  and denote the quotient,  $E/N$ , by  $G$ . Given a set of generators, written in normal form relative to  $G$  and  $N$ , of a subgroup,  $H \leq E$ , it is required to compute the following information relating to  $H$ :

- (i) The normal polycyclic subgroup,  $N_H = H \cap N$ , of  $H$ .
- (ii) A base,  $B_H$ , and strong generating set,  $T$ , relative to  $B_H$ , for the quotient,  $G_H = H/N_H \cong HN/N$ .
- (iii) Elements of  $E$  representing images of  $T$  in a left transversal,  $L_H$ , of  $N_H$  in  $H$ .
- (iv) Conjugates of each polycyclic generator of  $N_H$  by the images of  $T$  in  $L_H$ .
- (v) Data corresponding to Equations (3.1) and (3.2) in the context of  $H$ .

Items (iv) and (v) are calculated in a manner similar to that of the parent group; the details of the respective calculations are given in Subsection 4.2.2.

The difficulty lies in computing (i) and (iii). The approach taken is to extend the well-known Schreier–Sims method described in Subsection 2.1.3 of Chapter 2, to efficiently compute the segments of data described in (i), (ii) and (iii) simultaneously.

The newly developed version of the Schreier–Sims method, which forms the core of the subgroup generation procedure, is discussed in the next subsection.

In light of the application of the Second Isomorphism Theorem in (ii), the quotient,  $G_H$  may be regarded as a subgroup of the quotient,  $G$ , for which there is a known faithful permutation representation, thus facilitating computation of the required base and strong generating set. For the rest of this chapter,  $G_H$  will be identified with an isomorphic copy in  $HN/N$ .

## 4.2 Construction

The technical details of subgroup construction are discussed in this section.

### 4.2.1 The Extended Schreier–Sims Algorithm

The extended version of the Schreier–Sims algorithm is described in detail here. Let  $E$ ,  $N$  and  $G$  be as in Section 4.1. Given a set of elements of  $E$  that generate a subgroup,  $H \leq E$ , the extended Schreier–Sims algorithm aims to compute the intersection  $H \cap N$ , together with a base for the quotient,  $HN/N$ , and a set of elements of  $E$  (or more precisely  $HN$ ) whose images in  $HN/N$  define a strong generating set relative to the computed base.

#### The Permutation Action of the Polycyclic-by-Finite Group

Manipulating the elements of  $E$  relative to the (not faithful) permutation action of  $E$  induced by that of the quotient,  $G$ , is central to the operation of the extended Schreier–Sims algorithm.

Denote by  $\Omega$  the set of points on which  $G$  acts. Given an element,  $(g, n) \in E$ , and a point,  $\omega \in \Omega$ , one may unambiguously speak of the action of  $(g, n)$  on  $\omega$ , with obvious meaning, viz.

$$\omega^{(g,n)} = \omega^g.$$

The fundamental algorithms for orbit computation can be applied without modification in the case of this permutation action. Specifically, it is possible, using the methods described in Subsection 2.1.2 of Chapter 2, to compute the orbit of a point,  $\omega \in \Omega$ , under a set,  $X \subseteq E$ , of elements along with a Schreier vector encoding a transversal of the stabiliser,  $\langle X \rangle_\omega$ , in  $\langle X \rangle$ . It is important to note here that the normal subgroup component of each element encoded by such a Schreier vector is not truncated, i.e. the transversal elements are expressed as products over  $X$ , not over the image of  $X$  in  $G$ .

The definitions given in Subsection 2.1.3 of Chapter 2 for a base and a strong generating set may be reformulated to fit the context of this class of groups.

A sequence,  $B_H = (\gamma_1, \dots, \gamma_k)$ , of elements belonging to  $\Omega$  is called a *base* for the subgroup,  $H$ , of  $E$  if every element of  $H$  that fixes  $B_H$  pointwise belongs to  $H \cap N$ . The sequence,  $B_H$ , defines a stabiliser subgroup chain

$$H = H^{[1]} \geq H^{[2]} \geq \dots \geq H^{[k]} \geq H^{[k+1]} = H \cap N \quad (4.1)$$

where  $H^{[i]} = H_{(\gamma_1, \dots, \gamma_{i-1})}$  ( $i > 1$ ) is the pointwise stabiliser of  $\{\gamma_1, \dots, \gamma_{i-1}\}$ . The base,  $B_H$ , is called *non-redundant* if  $H^{[i+1]} < H^{[i]}$  for all  $i = 1, \dots, k$ . The orbits,  $\gamma_i^{H^{[i]}}$ , are called the *basic orbits* or *fundamental orbits* of  $H$  (relative to  $B_H$ ).

A *strong generating set* for  $H$  relative to  $B_H$  is a generating set,  $S_H$ , for  $H$  with the property that

$$\langle S_H \cap H^{[i]} \rangle = H^{[i]} \quad (4.2)$$

for  $i = 1, \dots, k + 1$ .

In the language of the definitions made above, the objective of the extended Schreier–Sims method may be stated more simply: given a set of generators of a subgroup,  $H \leq E$ , the method attempts to compute a base and a strong generating set for  $H$ .

Before turning to the central problem of how a base and a strong generating set

for a subgroup of a polycyclic-by-finite group are constructed, it shall be assumed for the moment that these are given, and a version of the sifting procedure for polycyclic-by-finite groups shall be introduced.

Let  $S_H$  be a strong generating set for the subgroup  $H$ , with associated base,  $B_H = (\gamma_1, \dots, \gamma_k)$ . It is assumed that the basic orbits,  $\Theta_i = \gamma_i^{H^{[i]}}$ , have been calculated and that Schreier vectors encoding transversals,  $R_i$ , of  $H^{[i+1]}$  in  $H^{[i]}$  exist for each  $i$ , with transversal elements at level  $i$  represented as products over  $S_H \cap H^{[i]}$ .

The sifting algorithm for polycyclic-by-finite groups operates as follows. Given  $(g, n) \in E$ , the algorithm attempts to find the coset representative  $(y_1, n'_1) \in R_1$  such that  $\gamma_1^{(g, n)} = \gamma_1^{(y_1, n'_1)}$ , and computes  $(g_2, n_2) = (g, n)(y_1, n'_1)^{-1} \in H^{[2]}$  if a coset representative is found. If no such coset representative exists, then  $(g, n)$  takes  $\gamma_1$  out of the orbit,  $\Theta_1$ , and so the algorithm breaks and returns the siftee,  $(g, n)$ , along with an integer indicating the level at which the break occurred, in this case 1. Otherwise, the algorithm continues and attempts find  $(y_2, n'_2) \in R_2$  such that  $\gamma_2^{(g_2, n_2)} = \gamma_2^{(y_2, n'_2)}$  and then computes  $(g_3, n_3) = (g_2, n_2)(y_2, n'_2)^{-1}$  if possible. The algorithm attempts to perform  $k$  iterations of this type, immediately breaking if, at any level, the base point is taken out of orbit, in which case the siftee and an integer indicating the level is returned. If the algorithm is able to perform  $k$  iterations successfully, then it returns the siftee and the integer,  $k + 1$ .

**Remark 4.1.** Note that, if  $(g, n) \in H$ , then the sifting algorithm will perform  $k$  iterations, and the siftee returned will have trivial first component, but possibly non-trivial second component. The element,  $(g, n) \in E$ , belongs to  $H$  if and only if the siftee returned after  $k$  iterations has a trivial first component and second component belonging to  $H \cap N$ . Thus, the sifting algorithm has reduced membership testing in  $H$  to membership testing in  $H \cap N$ , which can be accomplished using the already available methods for polycyclic groups.

The sifting procedure is presented in Algorithm 4.1. The function takes as input

an element,  $(g, n)$ , of a polycyclic-by-finite group,  $E$ , a base,  $B_H$ , for a subgroup,  $H \leq E$ , the sequence,  $\Theta = (\Theta_1, \dots, \Theta_k)$ , of basic orbits and a sequence,  $R = (R_1, \dots, R_k)$ , where  $R_i$  is a transversal of  $H^{[i+1]}$  in  $H^{[i]}$ . The method operates as described above to sift  $(g, n)$  in the hierarchy defined by  $B_H$ .

As in Subsection 2.1.3, sifting involves constructing exactly one transversal element for each base point, and thus requires  $O(d \log_2 |G|)$  multiplications in  $E$ , where  $d$  is the degree of the permutation representation for  $G$ .

---

**Algorithm 4.1** Sifting

---

```

1: function SIFT( $(g, n)$ ,  $B_H$ ,  $\Theta$ ,  $R$ )
2:    $(g_s, n_s) \leftarrow (g, n)$ 
3:   for  $i \leftarrow 1$  to  $k$  do                                      $\triangleright (g_s, n_s)$  fixes base points  $\gamma_1, \dots, \gamma_{i-1}$ 
4:      $\omega \leftarrow \gamma_i^{(g_s, n_s)}$ 
5:     if  $\omega \notin \Theta_i$  then
6:       return  $(g_s, n_s)$ ,  $i$ 
7:     end if
8:     Find the coset representative,  $(y_i, n'_i) \in R_i$ , such that  $\gamma_i^{(y_i, n'_i)} = \omega$ 
9:      $(g_s, n_s) \leftarrow (g_s, n_s) \cdot (y_i, n'_i)^{-1}$ 
10:  end for
11:  return  $(g_s, n_s)$ ,  $k + 1$ 
12: end function

```

---

### Constructing the Base and Strong Generating Set

Three modified versions of the Schreier–Sims algorithm are presented, each of which, when given a list of generators of a subgroup of a polycyclic-by-finite group, computes a base and strong generating set for that subgroup. The first operates in a manner similar to that described in Subsection 2.1.3 of Chapter 2. The second is a refinement of the first, with changes made in the method by which the normal subgroup is calculated. The third version focuses on the situation where it is required to use the base of the parent group in the representation of the subgroup.

The operation of the extended Schreier–Sims algorithm is based on the following reformulation of Lemma 2.5.



**Lemma 4.2.** *Let  $E$  be a polycyclic-by-finite group,  $H \leq E$  and let  $N \triangleleft E$  be polycyclic of finite index in  $E$ . Assume that there is a known permutation representation for the quotient  $G = E/N$ , and that the elements of  $E$  are expressed in normal form relative to  $G$  and  $N$ . Denote the set on which  $G$  acts by  $\Omega$  and let  $\{\gamma_1, \dots, \gamma_k\} \subseteq \Omega$ . For each  $i$  in  $\{1, \dots, k+1\}$ , let  $S_{H,j} \subseteq H_{(\gamma_1, \dots, \gamma_{j-1})}$  such that  $\langle S_{H,j} \rangle \geq \langle S_{H,j+1} \rangle$  holds for  $j \leq k$ . If  $H = \langle S_{H,1} \rangle$ ,  $S_{H,k+1} \subseteq H \cap N$ , and*

$$\langle S_{H,j} \rangle_{\gamma_j} = \langle S_{H,j+1} \rangle \quad (4.3)$$

*holds for each  $j$ , then  $B_H = (\gamma_1, \dots, \gamma_k)$  is a base for  $H$  and  $S_H = \bigcup_{j=1}^{k+1} S_{H,j}$  is a strong generating set for  $H$  relative to  $B_H$ .*

*Proof.* Induction on  $k$  is used here. The case  $k = 1$  is trivial, for  $\bigcup_{j=1}^1 S_{H,j}$  clearly fulfils the requirements of a strong generating set (relative to the base containing the single point  $\gamma_1$ ) for the group,  $\langle S_{H,1} \rangle$ .

Assume that the result holds for bases of length  $k-1$ . Then, in particular,  $S_H^* = \bigcup_{j=2}^k S_{H,j}$  is a strong generating set for  $\langle S_{H,2} \rangle$  relative to the base  $B_H^* = (\gamma_2, \dots, \gamma_k)$ .

Let  $H^{[i]} = H_{(\gamma_1, \dots, \gamma_{i-1})}$  for  $i = 2, \dots, k+1$ . To prove the lemma, it is required to verify that Equation (4.2) holds for each  $i$ . Setting  $j = 1$  in Equation (4.3) yields

$$H_{\gamma_1} = \langle S_{H,1} \rangle_{\gamma_1} = \langle S_{H,2} \rangle,$$

which implies the containment,

$$H_{\gamma_1} \leq \langle S_H \cap H_{\gamma_1} \rangle.$$

The reverse inclusion is obvious, thus

$$\langle S_H \cap H^{[2]} \rangle = H^{[2]},$$

and Equation (4.2) is satisfied for  $i = 2$ .

For  $i > 2$ , the inductive hypothesis implies that  $S_H^* \cap H_{(\gamma_1, \dots, \gamma_{i-1})}$  generates  $\langle S_{H,2} \rangle_{(\gamma_1, \dots, \gamma_{i-1})}$ , and so  $H^{[i]} \geq \langle S_H \cap H_{(\gamma_1, \dots, \gamma_{i-1})} \rangle \geq \langle S_H^* \cap H_{(\gamma_1, \dots, \gamma_{i-1})} \rangle = \langle S_{H,2} \rangle_{(\gamma_1, \dots, \gamma_{i-1})} = (H_{\gamma_1})_{(\gamma_1, \dots, \gamma_{i-1})} = H^{[i]}$ . Therefore,

$$\langle S_H \cap H^{[i]} \rangle = H^{[i]}$$

for each  $i$ , as required.  $\square$

Given a set,  $X$ , of generators for a subgroup,  $H \leq E$ , the extended Schreier–Sims algorithm constructs a base and strong generating set in the following way. A data structure containing a list,  $B_H = (\gamma_1, \dots, \gamma_k)$ , of already known elements of a non-redundant base is maintained, along with an approximation,  $S_{H,i}$ , for a generator set of the stabiliser,  $H_{(\gamma_1, \dots, \gamma_{i-1})}$ , for each  $i \in \{1, \dots, k+1\}$ . Throughout execution, the  $S_{H,i}$  satisfy the property that, for all  $i$ ,  $\langle S_{H,i} \rangle \geq \langle S_{H,i+1} \rangle$ , and  $S_{H,k+1} \subseteq H \cap N$ . The data structure is said to be *up-to-date below level  $j$*  if Equation (4.3) holds for each  $i$  in the range  $j < i \leq k$ .

In the case where the data structure is up-to-date below level  $j$ , a transversal,  $R_j$ , of  $\langle S_{H,j} \rangle_{\gamma_j}$  in  $\langle S_{H,j} \rangle$  is computed. Then a check is made to determine whether Equation (4.3) is satisfied for  $i = j$ . By Lemma 2.4, this can be done by sifting the Schreier generators obtained from  $R_j$  and  $S_{H,j}$  in the group,  $\langle S_{H,j+1} \rangle$ . By Remark 4.1, membership testing is possible in the group,  $\langle S_{H,j+1} \rangle$ , since Lemma 4.2 implies that  $\bigcup_{i=j+1}^{k+1} S_{H,i}$  is a strong generating set for  $\langle S_{H,j+1} \rangle$  relative to the base,  $(\gamma_{j+1}, \dots, \gamma_k)$ . If all Schreier generators are in  $\langle S_{H,j+1} \rangle$  then the data structure is up-to-date below level  $j - 1$ . Otherwise there exists a non-trivial siftee,  $(g_s, n_s)$ , at level  $t$  say. If  $t = k + 1$  and  $g_s \neq 1_G$ , then a new base point,  $\gamma_{k+1}$ , is appended to  $B_H$  from  $\text{supp}((g_s, n_s))$  and the set  $S_{H,k+2}$  is initialised with the contents of  $S_{H,k+1}$ . The siftee,  $(g_s, n_s)$ , is added to the sets  $S_{H,j+1}, \dots, S_{H,t}$ , and the data structure is now up-to-date below level  $\min(t, |B_H|)$ .

The algorithm initialises  $B_H$  to contain a sequence of points  $\gamma_1, \dots, \gamma_k$  in  $\Omega$  such that each point is moved by at least one generator in  $X$  with non-trivial first component, and sets  $S_{H,i}$  to  $X \cap \langle X \rangle_{(\gamma_1, \dots, \gamma_{i-1})}$  for  $i = 1, \dots, k+1$ . At that moment, the data structure is up-to-date below level  $k$ ; the algorithm terminates when the data structure becomes up-to-date below level 0.

The first version of the extended Schreier–Sims method is presented in Algorithm 4.2. The function takes as input a set,  $X$ , of elements of a polycyclic-by-finite group,  $E$ , and operates as described above to compute a base and strong generating set for  $H = \langle X \rangle$ . The notation of this subsection is used in the description of this algorithm (and subsequent versions); in particular, it is assumed that the elements of  $E$  are written in normal form relative to a normal polycyclic subgroup,  $N$ , with associated quotient,  $G$ , acting on a set,  $\Omega$ . The algorithm also assumes the existence of a procedure `APPEND` which, when supplied with an array (or sequence) and an element, appends the element to the end of the array (or sequence).

The normal subgroup,  $H \cap N$ , of  $H$  can be computed easily from the strong generating set  $S_H$  returned by `SCHREIERSIMS` as it is generated by the elements in  $S_H$  with trivial first component.

The most expensive computations in `SCHREIERSIMS` are the multiplications in  $E$  using its representation as an extension of  $N$  — from a time complexity perspective, these computations dominate those performed in  $G$  and in  $N$  (see Theorem 3.5 for a complexity analysis of the multiplication operation in  $E$ ). An estimate of the number of multiplications in  $E$  performed by `SCHREIERSIMS` is given in Theorem 4.3.

**Theorem 4.3.** *The extended Schreier–Sims algorithm is correct and requires  $O(d^2(\log_2|G|)^3 + |X|(d\log_2|G|)^2)$  multiplications in the polycyclic-by-finite group, where  $X$  is the generating set provided,  $G$  is the finite quotient used in the representation of the polycyclic-by-finite group, and  $d$  is the degree of the permutation representation of  $G$ .*

---

**Algorithm 4.2** Standard version of the extended Schreier–Sims algorithm

---

```

1: function SCHREIERSIMS( $X$ )
2:    $B_H \leftarrow ()$ ,  $k \leftarrow 0$ 
3:   for  $(g, n) \in X$  do
4:     if  $g \neq 1_G$  and  $(g, n) \in H_{(\gamma_1, \dots, \gamma_k)}$  then       $\triangleright (g, n)$  fixes all points in  $B_H$ 
5:       Find  $\gamma_{k+1} \in \Omega$  with  $\gamma_{k+1}^{(g,n)} \neq \gamma_{k+1}$ 
6:       APPEND( $B_H$ ,  $\gamma_{k+1}$ )
7:        $k \leftarrow k + 1$ 
8:     end if
9:   end for
10:  for  $i \leftarrow 1$  to  $k$  do
11:     $S_{H,i} \leftarrow X \cap H_{(\gamma_1, \dots, \gamma_{i-1})}$ ,  $H^{[i]} \leftarrow \langle S_{H,i} \rangle$ ,  $\Theta_i \leftarrow \gamma_i^{H^{[i]}}$ ,
12:    Compute a stabiliser transversal,  $R_i$ , in  $H^{[i]}$  corresponding to the orbit,
     $\Theta_i$ 
13:  end for
14:   $\Theta \leftarrow (\Theta_1, \dots, \Theta_k)$ ,  $R \leftarrow (R_1, \dots, R_k)$ 
15:   $S_{H,k+1} \leftarrow \{(g, n) \in X \mid g = 1_G\}$ 
16:   $N_H \leftarrow \langle S_{H,k+1} \rangle$        $\triangleright$  This subgroup is generated in the polycyclic group  $N$ 
17:   $i \leftarrow k$ 
18:  while  $i \geq 1$  do
19:    for  $\theta \in \Theta_i$  do
20:      Find the coset representative,  $(g, n) \in R_i$ , such that  $\gamma_i^{(g,n)} = \theta$ 
21:      for  $(x, n_x) \in S_{H,i}$  do
22:        Find the coset representative,  $(g', n') \in R_i$ , such that  $\gamma_i^{(g',n')} =$ 
         $\theta^{(x, n_x)}$ 
23:         $(g, n) \leftarrow (g, n) \cdot (x, n_x) \cdot (g', n')^{-1}$ 
24:        if  $(g, n) = (1_G, 1_N)$  then
25:          continue  $(x, n_x)$ 
26:        end if
27:       $uptodate \leftarrow \mathbf{true}$ 
28:       $(g, n), j \leftarrow \text{SIFT}((g, n), B_H, \Theta, R)$ 

```

---

---

```

29:         if  $g \neq 1_G$  then
30:              $uptodate \leftarrow \text{false}$ 
31:             if  $j > k$  then
32:                 Find  $\gamma_{k+1} \in \Omega$  with  $\gamma_{k+1}^{(g,n)} \neq \gamma_{k+1}$ 
33:                 APPEND( $B_H, \gamma_{k+1}$ ) ▷ Extend base
34:                  $k \leftarrow k + 1$ 
35:                  $S_{H,k+1} \leftarrow S_{H,k}$  ▷ Maintain inclusion
36:             end if
37:             else if  $n \notin N_H$  then
38:                  $uptodate \leftarrow \text{false}$ 
39:                  $S_{H,k} \leftarrow S_{H,k} \cup \{(g, n)\}$ 
40:                  $N_H \leftarrow \langle N_H, n \rangle$ 
41:                  $j \leftarrow k$ 
42:             end if
43:             if not  $uptodate$  then
44:                 for  $t \leftarrow i + 1$  to  $j$  do
45:                      $S_{H,t} \leftarrow S_{H,t} \cup \{(g, n)\}, H^{[t]} \leftarrow \langle S_{H,t} \rangle, \Theta_t \leftarrow \gamma_t^{H^{[t]}}$ ,
46:                     Compute a stabiliser transversal,  $R_t$ , in  $H^{[t]}$  corresponding
to the orbit,  $\Theta_t$ 
47:                 end for
48:                  $\Theta \leftarrow (\Theta_1, \dots, \Theta_k), R \leftarrow (R_1, \dots, R_k)$ 
49:                  $i \leftarrow j + 1$ 
50:                 break  $\theta$ 
51:             end if
52:         end for
53:     end for
54:      $i \leftarrow i - 1$ 
55: end while
56: return  $B_H, \bigcup_{i=1}^{k+1} S_{H,i}$ 
57: end function

```

---

*Proof.* Lemma 4.2 implies correctness.

The number of multiplications in  $E$  may be estimated by counting the number of times that SIFT is executed (Line 28). Since each Schreier generator is sifted exactly once, the number of times SIFT is executed is equal to the total number of Schreier generators.

Keeping the notation of SCHREIERSIMS, after initialisation in Lines 3–13, each  $R_i$  has size at most  $d$ , each  $S_{H,i}$  has size at most  $|X|$ , and,  $k$  is bounded by  $\log_2 |G|$ . Thus, the number of Schreier generators before any sift operations are performed is  $O(|X|d \log_2 |G|)$ .

Observe that although the sets  $R_i$  and  $S_{H,i}$  change during the operation of the algorithm, they are always only augmented and therefore any elements of  $R_i$  and  $S_{H,i}$  must be combined to a Schreier generator only once. The number of base points is at most  $\log_2|G|$ . For a fixed base point  $\gamma_i$ , the set,  $S_{H,i}$ , changes at most  $\log_2|G|$  times during the algorithm, since the group,  $\langle S_{H,i} \rangle$ , must increase each time an element is added to  $S_{H,i}$ . Combining this with the bound  $|R_i| \leq d$ , it follows that the number of Schreier generators constructed (Line 23) throughout the operation of the algorithm is  $O(d(\log_2|G|)^2)$ . The total number of Schreier generators is thus  $O(d(\log_2|G|)^2 + |X|d\log_2|G|)$ .

Each Schreier generator is sifted once and each sift requires  $O(d\log_2|G|)$  multiplications in the polycyclic-by-finite group. Therefore, the total number of multiplications that must be performed in the polycyclic-by-finite group is  $O(d^2(\log_2|G|)^3 + |X|(d\log_2|G|)^2)$ .  $\square$

Consider Lines 19–23 of SCHREIERSIMS. If the generator,  $(x, n_x)$ , has trivial first component, i.e.  $x = 1_G$ , then  $\theta^{(x, n_x)} = \theta$  whence  $(g', n') = (g, n)$  and the multiplication of elements in Line 23 is reduced to the conjugation,

$$(g, n) \cdot (1_G, n_x) \cdot (g, n)^{-1},$$

of a generator in  $N$  by an element of  $H$ . The algorithm proceeds to sift this conjugate down to level  $k + 1$ , adding it to the generator set of  $H \cap N$ .

In light of this, one may modify SCHREIERSIMS so that strong generators with trivial first component are not added at every level. Instead, generators of this type are kept in a separate set and a normal closure computation is performed after the structure becomes up-to-date below level 0. Eliminating strong generators with trivial first component from the data structure has the obvious advantage of speeding up both the orbit computations and the membership checks at each level.

A version of the extended Schreier–Sims algorithm incorporating these changes

is presented in Algorithm 4.3. The algorithm assumes the existence of a function `GENERATORS`, which, when supplied with a polycyclic group, returns a set of generators for that group.

The discussion above justifies the following remark, which implies the correctness of `NORMALCLOSURESCHREIERSIMS`.

**Remark 4.4.** The subgroup,  $H \cap N$ , is the normal closure in  $\langle X \rangle$  of the siftees appearing `NORMALCLOSURESCHREIERSIMS` with trivial first component.

The normal closure computation in Line 48 of `NORMALCLOSURESCHREIERSIMS` is performed using the standard method of repetitive adding of conjugates to a generating set. The available methods for polycyclic groups are used for membership testing of conjugates, and conjugation is of course performed using the multiplication method of Chapter 3.

From a theoretical perspective, the time complexity of this version of the extended Schreier–Sims method is no different from its predecessor. However, in practice, methods used to compute normal closures often terminate rapidly, and so this version may offer a slight advantage in efficiency.

Both `SCHREIERSIMS` and `NORMALCLOSURESCHREIERSIMS` attempt to compute a new base for the subgroup  $H$ . Situations may arise where one wishes to use the base of the parent group in the representation of the subgroup. A version of the extended Schreier–Sims algorithm with this minor simplification is presented in Algorithm 4.4. The function takes as input a set,  $X$ , of elements of a polycyclic-by-finite group,  $E$ , along with the base,  $B = (\gamma_1, \dots, \gamma_k)$ , of  $E$ .

As indicated in the discussion immediately following Theorem 3.5, smaller bases are preferable in relation to the performance of `MULTIPLY`. Taking this into consideration, it may be prudent in most instances to opt for a possible reduction in base size, as in Algorithms 4.2 and 4.3, unless the application specifically requires that the original base be retained.

**Algorithm 4.3** Normal closure version of the extended Schreier–Sims algorithm

---

```

1: function NORMALCLOSURESCHREIERSIMS( $X$ )
2:    $X_1 \leftarrow \{(g, n) \in X \mid g \neq 1_G\}$ ,  $X_2 \leftarrow \{(g, n) \in X \mid g = 1_G\}$      $\triangleright X = X_1 \cup X_2$ 
3:    $B_H \leftarrow ()$ ,  $k \leftarrow 0$ 
4:   for  $(g, n) \in X_1$  do
5:     if  $(g, n) \in H_{(\gamma_1, \dots, \gamma_k)}$  then                                 $\triangleright (g, n)$  fixes all points in  $B_H$ 
6:       Find  $\gamma_{k+1} \in \Omega$  with  $\gamma_{k+1}^{(g, n)} \neq \gamma_{k+1}$ 
7:       APPEND( $B_H$ ,  $\gamma_{k+1}$ )
8:        $k \leftarrow k + 1$ 
9:     end if
10:  end for
11:  for  $i \leftarrow 1$  to  $k$  do
12:     $S_{H,i} \leftarrow X_1 \cap H_{(\gamma_1, \dots, \gamma_{i-1})}$ ,  $H^{[i]} \leftarrow \langle S_{H,i} \rangle$ ,  $\Theta_i \leftarrow \gamma_i^{H^{[i]}}$ ,
13:    Compute a stabiliser transversal,  $R_i$ , in  $H^{[i]}$  corresponding to the orbit,
     $\Theta_i$ 
14:  end for
15:   $\Theta \leftarrow (\Theta_1, \dots, \Theta_k)$ ,  $R \leftarrow (R_1, \dots, R_k)$ 
16:   $i \leftarrow k$ 
17:  while  $i \geq 1$  do
18:    for  $\theta \in \Theta_i$  do
19:      Find the coset representative,  $(g, n) \in R_i$ , such that  $\gamma_i^{(g, n)} = \theta$ 
20:      for  $(x, n_x) \in S_{H,i}$  do
21:        Find the coset representative,  $(g', n') \in R_i$ , such that  $\gamma_i^{(g', n')} =$ 
         $\theta^{(x, n_x)}$ 
22:         $(g, n) \leftarrow (g, n) \cdot (x, n_x) \cdot (g', n')^{-1}$ 
23:        if  $(g, n) = (1_G, 1_N)$  then
24:          continue  $(x, n_x)$ 
25:        end if
26:         $(g, n), j \leftarrow \text{SIFT}((g, n), B_H, \Theta, R)$ 
27:        if  $g \neq 1_G$  then
28:          if  $j > k$  then
29:            Find  $\gamma_{k+1} \in \Omega$  with  $\gamma_{k+1}^{(g, n)} \neq \gamma_{k+1}$ 
30:            APPEND( $B_H$ ,  $\gamma_{k+1}$ )                                 $\triangleright$  Extend base
31:             $k \leftarrow k + 1$ 
32:             $S_{H,k} \leftarrow \emptyset$ 
33:          end if
34:          for  $t \leftarrow i + 1$  to  $j$  do
35:             $S_{H,t} \leftarrow S_{H,t} \cup \{(g, n)\}$ ,  $H^{[t]} \leftarrow \langle S_{H,t} \rangle$ ,  $\Theta_t \leftarrow \gamma_t^{H^{[t]}}$ ,
36:            Compute a stabiliser transversal,  $R_t$ , in  $H^{[t]}$  corresponding
            to the orbit,  $\Theta_t$ 
37:          end for

```

---



---

```

38:       $\Theta \leftarrow (\Theta_1, \dots, \Theta_k), R \leftarrow (R_1, \dots, R_k)$ 
39:       $i \leftarrow j + 1$ 
40:      break  $\theta$ 
41:      else if  $n \neq 1_N$  then
42:         $X_2 \leftarrow X_2 \cup \{(g, n)\}$ 
43:      end if
44:    end for
45:  end for
46:     $i \leftarrow i - 1$ 
47:  end while
48:   $N_H \leftarrow \langle X_2 \rangle^{\langle X_1 \rangle}$ 
49:   $X_2 \leftarrow \text{GENERATORS}(N)$ 
50:  return  $B_H, (\bigcup_{i=1}^k S_{H,i}) \cup X_2$ 
51: end function

```

---

### 4.2.2 Data

The data set described in Subsection 3.2.2 of Chapter 3 can be computed readily from the output of the extended Schreier–Sims method. Keeping the notation of Subsection 4.2.1, let  $B_H$  be a base for  $H$  and let  $S_H$  be a strong generating set for  $H$  relative to  $B_H$ . The subgroup,  $N_H = H \cap N$ , of  $H$  is generated by the elements of  $S_H$  with trivial first component, and this computation may be performed using the standard methods available for polycyclic groups, possibly involving the definition of a new polycyclic presentation for  $N_H$ .

Let  $\overline{T} = \{(x_1, n_1), \dots, (x_m, n_m)\}$  be the set containing all elements of  $S_H$  with non-trivial first component. It is assumed here that  $x_i \neq x_j$  for  $i \neq j$ . The set,  $T = \{x_1, \dots, x_m\}$ , is a strong generating set for  $G_H \cong H/N_H$ , relative to the base,  $B_H$ . Thus, using the permutation representation of  $G$ , one may compute  $G_H = \langle T \rangle \leq G$  along with a set of basic orbits and stabilisers (and Schreier vectors encoding basic stabiliser transversals) relative to  $B_H$ . The tails of Equations (3.1) in  $G_H$  are defined relative to this set of stabiliser transversals, and are calculated within  $G_H$  as described in Subsection 3.2.2.

The crucial point to note at this stage of the subgroup construction is that the transversal,  $L_H$ , of  $N_H$  in  $H$  (analogous to the transversal,  $L$ , of  $N$  in  $E$ ) is

**Algorithm 4.4** Base-preserving version of the extended Schreier–Sims algorithm

---

```

1: function KEEPBASESCHREIERSIMS( $X, B$ )
2:    $X_1 \leftarrow \{(g, n) \in X \mid g \neq 1_G\}, X_2 \leftarrow \{(g, n) \in X \mid g = 1_G\} \quad \triangleright X = X_1 \cup X_2$ 
3:    $B_H \leftarrow B$ 
4:   for  $i \leftarrow 1$  to  $k$  do
5:      $S_{H,i} \leftarrow X_1 \cap H_{(\gamma_1, \dots, \gamma_{i-1})}, H^{[i]} \leftarrow \langle S_{H,i} \rangle, \Theta_i \leftarrow \gamma_i^{H^{[i]}}$ ,
6:     Compute a stabiliser transversal,  $R_i$ , in  $H^{[i]}$  corresponding to the orbit,
        $\Theta_i$ 
7:   end for
8:    $\Theta \leftarrow (\Theta_1, \dots, \Theta_k), R \leftarrow (R_1, \dots, R_k)$ 
9:    $i \leftarrow k$ 
10:  while  $i \geq 1$  do
11:    for  $\theta \in \Theta_i$  do
12:      Find the coset representative,  $(g, n) \in R_i$ , such that  $\gamma_i^{(g,n)} = \theta$ 
13:      for  $(x, n_x) \in S_{H,i}$  do
14:        Find the coset representative,  $(g', n') \in R_i$ , such that  $\gamma_i^{(g',n')} =$ 
           $\theta^{(x,n_x)}$ 
15:         $(g, n) \leftarrow (g, n) \cdot (x, n_x) \cdot (g', n')^{-1}$ 
16:        if  $(g, n) = (1_G, 1_N)$  then
17:          continue  $(x, n_x)$ 
18:        end if
19:         $(g, n), j \leftarrow \text{SIFT}((g, n), B_H, \Theta, R)$ 
20:        if  $j \leq k$  then
21:          for  $t \leftarrow i + 1$  to  $j$  do
22:             $S_{H,t} \leftarrow S_{H,t} \cup \{(g, n)\}, H^{[t]} \leftarrow \langle S_{H,t} \rangle, \Theta_t \leftarrow \gamma_t^{H^{[t]}}$ ,
23:            Compute a stabiliser transversal,  $R_t$ , in  $H^{[t]}$  corresponding
              to the orbit,  $\Theta_t$ 
24:          end for
25:           $\Theta \leftarrow (\Theta_1, \dots, \Theta_k), R \leftarrow (R_1, \dots, R_k)$ 
26:           $i \leftarrow j + 1$ 
27:          break  $\theta$ 
28:        else if  $n \neq 1_N$  then
29:           $X_2 \leftarrow X_2 \cup \{(g, n)\}$ 
30:        end if
31:      end for
32:    end for
33:     $i \leftarrow i - 1$ 
34:  end while
35:   $N_H \leftarrow X_2^{\langle X_1 \rangle}$ 
36:   $X_2 \leftarrow \text{GENERATORS}(N)$ 
37:  return  $B_H, (\bigcup_{i=1}^k S_{H,i}) \cup X_2$ 
38: end function

```

---

predefined by the strong generating set returned by the extended Schreier–Sims method. That is, the image,  $\overline{x_i}$ , of  $x_i \in T$  in  $L_H$  is  $(x_i, n_i)$ . As in Subsection 3.2.2,  $\overline{T^{-1}} = \{\overline{x^{-1}} \mid x \in T\}$  is defined by the equation:

$$\overline{x^{-1}} = \begin{cases} \overline{x} & \text{if } x^2 = 1, \\ \overline{x}^{-1} & \text{otherwise.} \end{cases}$$

The sets,  $\overline{T}$  and  $\overline{T^{-1}}$ , correspond to the sets,  $\hat{S}$  and  $\hat{S}^{-1}$ , respectively, defined in Subsection 3.2.2 of Chapter 3.

Equipped with this definition for  $L_H$ , the tail elements of the shift equations are calculated as described in Subsection 3.2.2 — all multiplications performed in  $E$  using the multiplication algorithm of Chapter 3.

Finally, the conjugates,

$$a^{\overline{x^{-1}}}, a^{\overline{x^{-1}}} \in N_H,$$

are computed and stored for each polycyclic generator,  $a$ , of  $N_H$ , and each  $x \in T$ , again with all multiplications performed in  $E$  using the multiplication algorithm of Chapter 3.

This concludes the construction, and the multiplication algorithm may now be applied to elements of  $H$  written in normal form relative to  $G_H$  and  $N_H$ .

### 4.2.3 Membership Testing

Remark 4.1 forms the basis of the membership testing algorithm. Keeping the notation of Subsection 4.2.1, to test membership of  $(g, n) \in E$  in the subgroup,  $H$ ,  $(g, n)$  is sifted relative to the base and strong generating set computed by the extended Schreier–Sims algorithm. If an element with non-trivial first component is returned, then  $(g, n) \notin H$ . Otherwise, a siftee of the form  $(1_G, n')$  is returned, and  $(g, n) \in H$  if and only if  $n' \in H \cap N = N_H$ . The membership test in  $N_H$  is performed

using the available methods for polycyclic groups, see (Holt et al., 2005, Sec. 8.3).

The dominant computation in a test for membership in a subgroup is the application of the sifting method, which requires  $O(d \log_2 |G|)$  multiplications in  $E$ , where  $d$  is the degree of the permutation representation for  $G$  (see Subsection 4.2.1).

## 4.3 Applications

Following immediately from the subgroup generation algorithm is a function which computes the normal closure of a subgroup generated by a finite set of elements. This function may then be used to design algorithms which find derived subgroups and commutator subgroups by computing the normal closure of a suitable set of commutators. Using these algorithms, one may write simple iterative procedures which construct derived and lower central series. For a review of how these standard functions are implemented, see (Holt et al., 2005, Sec. 3.3).

In the subsections below, two straightforward yet useful applications of subgroup generation are given. The methods presented heavily exploit the polycyclic-by-finite structure of the group in question. Standard notation is kept in this section, viz.  $E$  is a polycyclic-by-finite group represented as an extension of a polycyclic group,  $N$ , by the finite quotient,  $G$ . The natural map,  $E \rightarrow G$ , is denoted by  $\rho$ .

### 4.3.1 The Soluble Radical

For a group,  $E$ , the *soluble radical*, denoted by  $\mathbf{O}_\infty(E)$ , is defined as the largest soluble normal subgroup of  $E$ . This subsection contains the outline of a method which computes the soluble radical of the polycyclic-by-finite group,  $E$ .

It follows from the definition of the soluble radical that  $N \leq \mathbf{O}_\infty(E)$ . Thus,  $\mathbf{O}_\infty(E)$  is the preimage under  $\rho$  of  $\mathbf{O}_\infty(G)$ .

The permutation representation of  $G$  may be utilised to compute  $\mathbf{O}_\infty(G)$ ; for a detailed discussion of the soluble radical algorithm for permutation groups, see

(Seress, 2003, Chap. 6). Given a generating set,  $X$ , of  $\mathbf{O}_\infty(G)$ , the method described Subsection 3.3.1 can be used to find a set,  $\overline{X} \subseteq E$ , which, under  $\rho$ , maps to  $X$ . The soluble radical of  $E$  is then  $\langle \overline{X}, N \rangle$ .

### 4.3.2 Sylow Subgroups

In this subsection, it is assumed that  $E$  is finite.

#### Construction

Let  $p$  be a prime dividing  $|E|$  and let  $H_p$  be a Sylow  $p$ -subgroup of  $G$ . Observe that a Sylow  $p$ -subgroup of the preimage,  $\overline{H_p} = \rho^{-1}(H_p)$ , is a Sylow  $p$ -subgroup of  $E$ .

Thus, one may compute a Sylow  $p$ -subgroup of  $E$  as follows. First, find a Sylow  $p$ -subgroup,  $H_p$ , of  $G$ . This is done using the available methods for permutation groups (see Holt et al., 2005, Chap. 4). Then, using a method similar to that of Subsection 4.3.1, generate  $\overline{H_p} = \rho^{-1}(H_p)$  as a subgroup of  $E$ . This group is finite and soluble, and so, one may produce a consistent power-conjugate presentation defining it using the method of Subsection 3.3.4 of Chapter 3. The existing methods for computing Sylow subgroups in finite soluble groups represented by power-conjugate presentations are now applicable (see Seress, 2003, Chap. 7). Finally, the isomorphism from the subgroup defined by the power-conjugate presentation can be used to map a generating set for a Sylow  $p$ -subgroup back into  $E$ .

#### Conjugacy

It is also possible to design an algorithm which, when given two Sylow  $p$ -subgroups of  $E$ , returns an element which conjugates one Sylow subgroup to the other. The first step is to find, using the available functions for permutation groups (see Holt et al., 2005, Chap. 4), an element  $g \in G$  which conjugates  $\rho(P_1)$  to  $\rho(P_2)$  in  $G$ . Using the method of Subsection 3.3.1 a preimage,  $\overline{g} \in E$ , for  $g$  may be found. It follows

then that  $P_1^{\bar{g}}$  is a Sylow  $p$ -subgroup of the group,  $\rho^{-1}(\rho(P_2)) = P_2N$ , which contains  $P_2$  as a Sylow  $p$ -subgroup.

The group,  $P_2N$ , is finite and soluble, and so may be represented by a consistent power-conjugate presentation. One may then find, using the available methods for finite soluble groups (see Seress, 2003, Chap. 7), a conjugating element,  $y \in P_2N$ , such that  $P_1^{\bar{g}y} = P_2$ , as required.

# Chapter 5

## Conjugacy

The material presented in this chapter utilises the machinery developed thus far to perform some advanced structural calculations within polycyclic-by-finite groups. Specifically, the question of element conjugacy is addressed.

The chapter begins with an algorithm to compute the centre of a given polycyclic-by-finite group. Following this, a method to construct centralisers of elements belonging to a finite polycyclic-by-finite group is described. The centraliser algorithm is developed using the theory of extensions and cohomology; a synopsis of the required material is found in Section 2.3 of Chapter 2. The problem of determining whether two elements are conjugate is intimately connected to the determination of centralisers, and may be solved algorithmically with a slight modification of the centraliser method. This algorithm is presented as the final instalment of the chapter.

The underlying idea of the algorithms presented here is to transform the problem into one of linear algebra, and then use the available methods for matrix groups to perform the computation.

## 5.1 The Centre

The approach to computing the centre of a given polycyclic-by-finite group,  $E$ , is as follows:

1. Find a normal abelian subgroup,  $A$ , of  $E$  that contains  $\mathbf{Z}(E)$ .
2. For each member,  $x$ , of a generating set,  $X$ , for  $E$ , compute the matrix,  $M_x$ , that specifies the conjugation action of that generator on elements of  $A$  (written as vectors).
3. Construct a matrix whose null space corresponds to the image of  $Z(E)$  in  $A$ .
4. Map a set of vectors generating  $Z(E)$  in  $A$  back into  $E$ .

The following simple lemma furnishes one with a feasible choice for the normal abelian subgroup.

**Lemma 5.1.** *Let  $G$  be a group with soluble radical,  $S$ . Then  $\mathbf{Z}(G) \trianglelefteq \mathbf{Z}(S) \trianglelefteq G$ .*

*Proof.* The subgroup,  $\mathbf{Z}(G)S$ , is soluble, so  $\mathbf{Z}(G) \leq S$  and hence,  $\mathbf{Z}(G) \trianglelefteq \mathbf{Z}(S)$ . Normality of  $\mathbf{Z}(S)$  in  $G$  follows from  $\mathbf{Z}(S)$  being characteristic in  $S$ .  $\square$

Let  $S$  be the soluble radical of  $E$ . Then by Lemma 5.1,  $\mathbf{Z}(E)$  is contained in the normal abelian subgroup,  $\mathbf{Z}(S)$ .

The soluble radical,  $S$ , can be computed as a subgroup of  $E$  using the method described in Subsection 4.3.1 of Chapter 4, after which it is rewritten as a polycyclic group by the procedure given in Subsection 3.3.4 of Chapter 3. The available functions for polycyclic groups may then be utilised to compute the centre,  $\mathbf{Z}(S)$ , of  $S$  as an abelian group; see (Holt et al., 2005, Chap. 8) for the finite case and (Eick, 2001) for the infinite case.

Write  $\mathbf{Z}(S)$  as an abelian group,  $A$ , with invariant factor decomposition,

$$\mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_t} \times \mathbb{Z}^k,$$



where  $m_i \mid m_{i+1}$  for each  $i$ . Writing the elements of  $A$  as row vectors, let  $\vec{a}_1, \dots, \vec{a}_{t+k}$  be the standard basis for  $A$  relative to its invariant factor decomposition. Let  $x \in X$ . Regarding matrices as acting on the right, the  $i$ -th row of the matrix,  $M_x$ , over  $\mathbb{Z}$ , defining conjugation in  $A$  by  $x$  is given by

$$\vec{a}_i x$$

expressed as a row vector, where  $E$  acts on  $A$  by conjugation.

The element,  $\vec{a}_i x$ , is found by first mapping  $\vec{a}_i$  into the group  $E$ , and then using the multiplication algorithm of Chapter 3 to perform the conjugation. The result is mapped back into  $A$  and written as a row vector.

An element,  $\vec{z} \in A$ , lies in  $\mathbf{Z}(E)$  if and only if, for each  $x \in X$ ,

$$\vec{z} M_x = \vec{z} \tag{5.1}$$

A matrix whose null space corresponds to the set of all such  $z$  is constructed as follows. Let  $D$  be the  $t \times (t+k)$  matrix over  $\mathbb{Z}$ ,

$$\begin{pmatrix} m_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & m_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & m_t & 0 & \cdots & 0 \end{pmatrix}.$$

Let  $X = \{x_1, \dots, x_q\}$  and let  $T_x = M_x - I$  for each  $x \in X$ , where  $I$  is the identity  $(t+k) \times (t+k)$  matrix. Form the following matrix,  $Q$ , over  $\mathbb{Z}$

$$\begin{pmatrix} T_{x_1} & T_{x_2} & \cdots & T_{x_q} \\ D & & & \\ & D & & \\ & & \ddots & \\ & & & D \end{pmatrix},$$

where blank spaces indicate zero entries.

**Claim 5.2.** If a vector,  $\vec{z} \in A$ , lies in  $\mathbf{Z}(E)$ , then there exists a vector  $(\vec{z} \mid \vec{v}) \in \mathbb{Z}^{t(q+1)+k}$  in the null space of  $Q$ . Conversely, the first  $t+k$  entries of any vector in the null space of  $Q$  define a vector belonging to  $\mathbf{Z}(E)$ .

*Proof.* Suppose that  $\vec{z} \in A$  lies in  $\mathbf{Z}(E)$ . Then for each  $x_i \in X$ ,  $\vec{z}T_{x_i} = \vec{0}$ , equality being in the abelian group,  $A \cong \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_t} \times \mathbb{Z}^k$ . Viewing  $\vec{z}$  as a vector in  $\mathbb{Z}^{t+k}$ ,  $\vec{z}T_{x_i} = \vec{0}$  is equivalent to  $\vec{z}T_{x_i} = -\vec{v}_i D$  for some  $\vec{v}_i \in \mathbb{Z}^t$ . Hence, the vector  $(\vec{z} \mid \vec{v}_1 \mid \cdots \mid \vec{v}_q)$  belongs to  $\ker(Q)$ .

Conversely let  $\vec{v} \in \ker(Q)$  and partition  $\vec{v}$  into segments, the first of length  $t+k$ , and the rest each of length  $t$ :  $(\vec{z} \mid \vec{v}_1 \mid \cdots \mid \vec{v}_q)$ . Then  $\vec{z}T_{x_i} = -\vec{v}_i D$  for each  $i$ , which, as above, is equivalent to  $\vec{z}T_{x_i} = \vec{0}$  in  $A$ , whence  $\vec{z}$  lies in  $\mathbf{Z}(E)$ .  $\square$

Thus, the image of  $\mathbf{Z}(E)$  in  $A$ , can be recovered from a generating set of  $\ker(Q)$ . Mapping this image back into  $E$  completes the computation.

Pseudocode for computing the centre in this manner is presented in Algorithm 5.1. The function takes as input a polycyclic-by-finite group,  $E$ , generated by a set,  $X = \{x_1, \dots, x_q\}$ .

The runtime of CENTRE is dominated by the calculation of the null space in Line 9. The fastest known algorithms to perform such computations use  $p$ -adic expansions, and were introduced by Dixon (1982). They require  $O(c^4(\log_2 r)^2)$  time

---

**Algorithm 5.1** Finding the centre

---

```

1: function CENTRE( $E$ )
2:   Find the soluble radical,  $S$ , of  $E$ 
3:   Write  $Z(S)$  as an abelian group:  $Z(S) \stackrel{\phi}{\cong} \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_t} \times \mathbb{Z}^k$  with standard
   basis  $\{\vec{a}_1, \dots, \vec{a}_{t+k}\}$  relative to this decomposition
4:   for  $x \in X$  do
5:      $T_x \leftarrow \begin{pmatrix} \vec{a}_1 x \\ \vdots \\ \vec{a}_{t+k} x \end{pmatrix} - I_{t+k}$ 
6:   end for
7:    $D \leftarrow \begin{pmatrix} m_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & m_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & m_t & 0 & \cdots & 0 \end{pmatrix} \quad \triangleright D \text{ has dimension } t \times (t+k)$ 
8:    $Q \leftarrow \left( \begin{array}{c|c|c|c} T_{x_1} & T_{x_2} & \cdots & T_{x_q} \\ \hline D & & & \\ \hline & D & & \\ \hline & & \ddots & \\ \hline & & & D \end{array} \right)$ 
9:   Compute generating set,  $Y$ , for  $\ker(Q)$ 
10:   $Y_A \leftarrow \emptyset$ 
11:  for  $\vec{y} \in Y$  do
12:    Form vector,  $\vec{y}_A$  defined by the first  $t+k$  entries of  $\vec{y}$ 
13:     $Y_A \leftarrow Y_A \cup \{\vec{y}_A\}$ 
14:  end for
15:  return  $\langle \phi^{-1}(Y_A) \rangle$ 
16: end function

```

---

for an  $r \times c$  matrix. For recent developments in this area, see (Haramoto and Matsumoto, 2009). It is worth noting here that, if  $k = 0$  and  $m_i = m$  for each  $i$  and some  $m \in \mathbb{N}$ , then the computation can be performed entirely in  $\mathbb{Z}_m$ , offering a significant improvement in efficiency. For a practical account of computations of this type, the reader is referred to (Holt et al., 2005, Chap. 7).

## 5.2 The Conjugacy Problem

The Conjugacy Problem is one of the fundamental decision problems in group theory, and it is known that the problem is undecidable for many classes of groups (see Chandler and Magnus, 1982). The class of polycyclic-by-finite groups, however, is a subclass of the class of finitely presented groups that possess solvable conjugacy problem. For a systematic account of the decision problems that are solvable in the class of polycyclic-by-finite groups, the reader is referred to (Baumslag et al., 1991).

The treatment of the conjugacy problem in this section focuses on the finite case, and extends the ideas of Holt et al. (2005, Sec. 8.8). It may be possible to generalise the methods presented here to cope with infinite groups using the work of Eick (2001). However, at present, the algorithms are primarily targeted at finite insoluble groups with large soluble normal subgroup.

### 5.2.1 Centralisers

Let  $E$  be a finite group with soluble normal subgroup, and assume that  $E$  is represented as described in Chapter 3. In this subsection, a method of computing

$$C_E(e) = \{f \in E \mid fe = ef\}$$

for an element,  $e \in E$ , is introduced.

The centraliser,  $C_E(e)$ , is the stabiliser of  $e$  under the conjugation action of

$E$  on its elements. Therefore, it is possible, in theory, to design a deterministic algorithm to compute  $C_E(e)$  by adapting the orbit-stabiliser method described in Subsection 2.1.2 to this case. However, this approach would inevitably involve computing and storing explicitly the entire orbit,  $e^E$ . This can be time and space consuming. A more effective strategy is presented in the sequel.

### Affine Actions

Before describing the algorithm itself, a digression must be made to introduce the concept of an *affine action* of a group on a vector space, which is required for the application at hand.

**Definition 5.3.** Let  $V \cong K^d$  be a finite-dimensional vector space over a field  $K$ , let  $\varphi: G \rightarrow \text{End}_K(V)$  be a linear action of a group,  $G$ , on  $V$ , and let  $\delta: G \rightarrow V$  be a derivation (see Equation (2.6)) with respect to the action,  $\varphi$ . Then the map,  $\alpha: G \rightarrow \text{Sym}(V)$ , defined by  $v^{\alpha(g)} = v^{\varphi(g)} + \delta(g)$  is called an *affine action* of  $G$  on  $V$ .

Keeping the notation of Definition 5.3, it is straightforward to check that  $\alpha$  is a homomorphism:

$$\begin{aligned} v^{\alpha(gh)} &= v^{\varphi(gh)} + \delta(gh) \\ &= v^{\varphi(gh)} + \delta(g)^{\varphi(h)} + \delta(h) \\ &= (v^{\varphi(g)} + \delta(g))^{\varphi(h)} + \delta(h) \\ &= v^{\alpha(g)\alpha(h)}. \end{aligned}$$

Thus, an affine action is an action of a group,  $G$ , on a vector space,  $V$ , but not a linear action unless  $\delta = 0$ .

If  $V$  is identified with  $K^d$  and  $\text{End}_K(V)$  with  $\text{GL}(d, K)$ , then the affine action,

$\alpha$ , corresponds to the homomorphism,  $\hat{\alpha}: G \rightarrow \text{GL}(d+1, K)$ , given by

$$\hat{\alpha}(g) = \left( \begin{array}{c|c} \varphi(g) & \begin{smallmatrix} 0 \\ \vdots \\ 0 \end{smallmatrix} \\ \hline \delta(g) & 1 \end{array} \right),$$

where, for  $v \in K^n$ ,  $(v^{\alpha(g)} \mid 1) = (v \mid 1) \cdot \hat{\alpha}(g)$ .

### The Centraliser Algorithm

Assume that the group,  $E$ , is represented as described in Chapter 3 with respect to the normal subgroup,  $N$ , and associated quotient,  $G$ . The normal subgroup,  $N$ , is soluble and finite, and hence possesses a normal elementary abelian series:

$$N = N_1 \supseteq \cdots \supseteq N_t \supseteq N_{t+1} = 1.$$

The basic idea of the method is an induction downwards along this normal series. Suppose that  $N_t$  is a  $p$ -group of rank,  $d$ , say, where  $p$  is prime, and assume, by induction, that the centraliser in the factor group  $E/N_t$  has already been computed. That is, a set of generators in  $E$  whose image in  $E/N_t$  generates  $\mathbf{C}_{E/N_t}(eN_t)$ , is known. Define  $C$  to be the subgroup of  $E$  such that  $C/N_t = \mathbf{C}_{E/N_t}(eN_t)$ . The following proposition is elementary.

**Proposition 5.4.** *The mapping  $\delta: C \rightarrow N_t$  defined by  $c \mapsto [e, c]$  is a derivation.*

*Proof.* For  $e \in E$  and  $c_1, c_2 \in C$ :

$$[e, c_1 c_2] = e^{-1} (c_1 c_2)^{-1} e c_1 c_2 = e^{-1} c_2^{-1} e [e, c_1] c_2 = [e, c_1]^{c_2} [e, c_2].$$

The commutativity of  $N_t$  is used in the last equality. □

The group  $N_t$  may be identified with the additive group  $\mathbb{F}_p^d$ , and additive notation may be used for its elements. The conjugation action of  $C$  on  $N_t$  may then be written as a homomorphism,  $\varphi: C \rightarrow \text{GL}(d, \mathbb{F}_p)$ , and one may consider  $\delta$  as a derivation of the form  $\delta: C \rightarrow \mathbb{F}_p^d$ . These can be combined to give an affine action,  $\alpha: C \rightarrow \text{GL}(d+1, \mathbb{F}_p^d)$ , defined by:

$$c \mapsto \left( \begin{array}{c|c} \varphi(c) & \begin{smallmatrix} 0 \\ \vdots \\ 0 \end{smallmatrix} \\ \hline \delta(c) & 1 \end{array} \right).$$

This affine action yields the following characterisation of  $\mathbf{C}_E(e)$ .

**Proposition 5.5.** *The centraliser,  $C_E(e)$ , is the stabiliser in  $C$  of the vector,*

$$(0, \dots, 0, 1) \in \mathbb{F}_p^{d+1},$$

where  $C$  acts on  $\mathbb{F}_p^{d+1}$  via  $\alpha$ .

*Proof.* The stabiliser in  $C$  of  $(0, \dots, 0, 1)$  under the action of  $\alpha$  is the kernel of the derivation,  $\delta$ . By the definition of  $\delta$ ,  $\ker(\delta) = \mathbf{C}_C(e)$ . As  $\mathbf{C}_E(e) \leq C$ , it follows that  $\mathbf{C}_C(e) = \mathbf{C}_E(e)$ .  $\square$

Hence one obtains an effective method to compute  $\mathbf{C}_C(e)$ . The algorithm commences by computing a set of elements in  $E$  whose images in the quotient group,  $E/N$ , generate the centraliser,  $\mathbf{C}_{E/N}(e)$ . This centraliser is computed using the known permutation representation for  $E/N$  by the already available methods for permutation groups described in (Seress, 2003) and (Holt et al., 2005, Chap. 4). The algorithm then proceeds to iterate down the normal elementary abelian series, solving the problem for successively larger quotients.

This method is similar to the method involving a single orbit-stabiliser computation, but the induction method computes relatively small orbits of vectors instead of one relatively large orbit of elements in a polycyclic-by-finite group. Thus, the induction method is usually more efficient than the single orbit-stabiliser application.

An algorithm for computing the centralisers in this manner is presented in Algorithm 5.2. The function takes as input a polycyclic-by-finite group,  $E$ , and an element,  $e \in E$ , and operates as described above to compute  $\mathbf{C}_E(e)$ . The notation of this subsection is kept in the description of the algorithm. In particular, it is assumed that the group,  $E$ , is represented as described in Chapter 3 with respect to the normal subgroup,  $N$ , and associated quotient,  $G$ .

---

**Algorithm 5.2** Finding centralisers

---

```

1: function CENTRALISER( $E, e$ )
2:   Compute a set,  $X$ , of generators in  $E$  for the image of the centraliser,
    $\mathbf{C}_{E/N}(eN)$  in  $E$ 
3:   Compute a normal elementary abelian series,  $N = N_1 \supseteq \dots \supseteq N_t \supseteq N_{t+1} = 1$ , for  $N$ 
4:   for  $i \leftarrow 1$  to  $t$  do
5:     Express the elementary abelian section  $N_i/N_{i+1}$  as vector space,  $V$ , with
     standard basis  $\{\vec{v}_1, \dots, \vec{v}_d\}$ 
6:     for  $x \in X$  do
7:       Express  $\delta(x) = e^{-1}x^{-1}exN_{i+1}$  as a vector,  $\vec{u}_x \in V$ 
8:        $M_x \leftarrow \left( \begin{array}{c|c} \vec{v}_1x & 0 \\ \vdots & \vdots \\ \vec{v}_{t+k}x & 0 \\ \hline \vec{u}_x & 1 \end{array} \right) \quad \triangleright x \text{ acts on } N_i/N_{i+1} \text{ by conjugation}$ 
9:     end for
10:    Compute the stabiliser,  $B_{\vec{w}}$ , of  $\vec{w} = (0, \dots, 0, 1)$  in  $B = \langle M_x \mid x \in X \rangle$ 
11:    Find the image,  $X'$ , in  $E$  of a generating set for  $B_{\vec{w}}$ 
12:     $X \leftarrow X'$ 
13:  end for
14:  return  $\langle X \rangle$ 
15: end function

```

---



### 5.2.2 Conjugacy Testing

The problem of checking whether two elements  $e, f \in E$  are conjugate can be solved with a variation of the centraliser algorithm.

Keeping the notation of Subsection 5.2.1, assume, by induction, that the centraliser in the factor group  $E/N_t$  has already been computed. That is, a set of generators in  $E$  whose image in  $E/N_t$  generates  $C/N_t = \mathbf{C}_{E/N_t}(eN_t)$ , is known. Assume also, again by induction, that  $e$  and  $f$  are equal in the factor group  $E/N_t$ . The following proposition forms the basis of the conjugacy testing algorithm.

**Proposition 5.6.** *If the elements  $e, f \in E$  are conjugate, then the element induced by  $e^{-1}f$  must lie in the orbit of the vector,*

$$(0, \dots, 0, 1) \in \mathbb{F}_p^{d+1},$$

under  $C$ , where  $C$  acts on  $\mathbb{F}_p^{d+1}$  via  $\alpha$ .

*Proof.* Suppose that  $e$  and  $f$  are conjugate in  $E$ . Then there exists  $y \in E$  such that  $e^y = f$ , or equivalently

$$[e, y] = e^{-1}f.$$

By assumption,  $e^{-1}f \in N_t$  and so, from the equality above,  $[e, y] \in N_t$ , or equivalently  $N_t y \in C/N_t$ .

Therefore,  $y \in C$ , and

$$\delta(y) = [e, y] = e^{-1}f,$$

and the element induced by  $e^{-1}f$  lies in the orbit of the vector,  $(0, \dots, 0, 1) \in \mathbb{F}_p^{d+1}$ , under  $C$ . □

Each time Line 10 of CENTRALISER is executed, a stabiliser computation is performed. In the test for conjugacy, the underlying orbit is also computed. By Proposition 5.6, if the element induced by  $e^{-1}f$  is not in this orbit, then  $e$  and  $f$

are not conjugate. Otherwise, there exists an element,  $y \in E$ , such that  $e^{-1}y^{-1}ey$  is equal to  $e^{-1}f$  in the considered factor, i.e.  $e^y$  is equal to  $f$  in the considered factor. This conjugating element is found using a transversal of the stabiliser in image of the affine map. The element,  $f$ , is modified to  $f^{y^{-1}}$ , so that  $e$  and  $f$  are now equal in the considered factor (as in Proposition 5.6), and the algorithm proceeds to the next induction step.

The algorithm commences by attempting to find an element conjugating  $e$  to  $f$  in the quotient group,  $E/N$  (see (Seress, 2003) and (Holt et al., 2005, Chap. 4) for details).

An algorithm for checking element conjugacy in this manner is presented in Algorithm 5.2. The function takes as input a polycyclic-by-finite group,  $E$ , and elements,  $e, f \in E$ , and operates as described above to determine whether  $e$  and  $f$  are conjugate in  $E$ , returning an element,  $y \in E$ , such that  $e^y = f$  in the affirmative case. The notation of this subsection is kept in the description of the algorithm. In particular, it is assumed that the group,  $E$ , is represented as described in Chapter 3 with respect to the normal subgroup,  $N$ , and associated quotient,  $G$ .

---

**Algorithm 5.3** Testing Element Conjugacy
 

---

```

1: function ISCONJUGATE( $E, e, f$ )
2:   if  $e$  and  $f$  are not conjugate in  $E/N$  then
3:     return false, -
4:   end if
5:   Find  $y \in E$  such that  $e^y$  is equal to  $f$  in  $E/N$ 
6:    $f \leftarrow f^{y^{-1}}$ 
7:   Compute a set,  $X$ , of generators in  $E$  for the image of the centraliser,
    $C_{E/N}(eN)$  in  $E$ 
8:   Compute a normal elementary abelian series,  $N = N_1 \supseteq \dots \supseteq N_t \supseteq N_{t+1} =$ 
    $1$ , for  $N$ 
9:   for  $i \leftarrow 1$  to  $t$  do
10:    Express the elementary abelian section  $N_i/N_{i+1}$  as vector space,  $V$ , with
    standard basis  $\{\vec{v}_1, \dots, \vec{v}_d\}$ 
11:    for  $x \in X$  do
12:      Express  $\delta(x) = e^{-1}x^{-1}exN_{i+1}$  as a vector,  $\vec{u}_x \in V$ 
13:      
$$M_x \leftarrow \left( \begin{array}{c|c} \vec{v}_1x & 0 \\ \vdots & \vdots \\ \vec{v}_{t+k}x & 0 \\ \hline \vec{u}_x & 1 \end{array} \right) \quad \triangleright x \text{ acts on } N_i/N_{i+1} \text{ by conjugation}$$

14:    end for
15:    Compute the orbit,  $O_{\vec{w}}$ , and stabiliser,  $B_{\vec{w}}$ , of  $\vec{w} = (0, \dots, 0, 1)$  in  $B =$ 
     $\langle M_x \mid x \in X \rangle$ 
16:    Express  $e^{-1}f$  as a vector,  $\vec{u}_{\text{test}} \in V$ 
17:    if  $\vec{u}_{\text{test}} \notin O_{\vec{w}}$  then
18:      return false, -
19:    end if
20:    Find  $y_{\text{local}} \in E$  such that  $e^{y_{\text{local}}}$  is equal to  $f$  in  $E/N_{i+1}$ 
21:     $f \leftarrow f^{y_{\text{local}}^{-1}}$   $\triangleright f$  is now equal to  $e$  in  $G/N_{i+1}$ 
22:     $y \leftarrow y_{\text{local}} \cdot y$ 
23:    Find the image,  $X'$ , in  $E$  of a generating set for  $B_{\vec{w}}$ 
24:     $X \leftarrow X'$ 
25:  end for
26:  return true,  $y$ 
27: end function

```

---

# Chapter 6

## Conclusion

This chapter begins by illustrating the algorithms developed in the thesis by performing some example computations on polycyclic-by-finite groups using the package developed. The following section discusses aspects of the implementation, including native support for polycyclic-by-finite groups in the Magma Computational Algebra System based on the work of this thesis. The final section outlines the potential for continued research in this direction.

### 6.1 Examples and Run-times

Presented here is a selection of examples of computation with polycyclic-by-finite groups using the code developed. See Appendix A for details on its usage. Tables of run-times for various computations in several different example groups are given at the end of this section.

The first three examples use groups belonging to the database of perfect groups provided by Magma.

**Example 6.1.** Element arithmetic and conjugacy testing in a perfect group.

```
> D := PerfectGroupDatabase();  
  
> E := PermutationGroup(D, 1920, 7);  
  
> Order(E);  
  
1920  
  
> N := SolubleRadical(E);  
  
> Q, rho := RadicalQuotient(E);  
  
> G := PCBFMasterConstruct(E, N, Q, rho); // construct group  
  
> PCBFGrpOrder(G); // order of group  
  
1920  
  
> g := PCBFRandom(G); // random element  
  
> h := PCBFRandom(G);  
  
> f := PCBFMult(G, g, h); // g * h  
  
> finv := PCBFInv(G, f); // f^-1  
  
> e := PCBFMult(G, f, finv);  
  
> PCBFIsIdentity(G, e); // e should be the identity  
  
true  
  
> c := PCBFConjugate(G, g, h); // h^-1 * g * h  
  
> isconj, hprime := PCBFIsConjugate(G, g, c); // check conjugacy - should be  
  
true  
  
> isconj;  
  
true
```

**Example 6.2.** Computing the centre of a perfect group.

```
> D := PerfectGroupDatabase();  
  
> E := PermutationGroup(D, 322560, 1);  
  
> N := SolubleRadical(E);  
  
> Q, rho := RadicalQuotient(E);  
  
> G := PCBFMasterConstruct(E, N, Q, rho);  
  
> PCBFGrpOrder(G);  
  
322560  
  
> Z := PCBFCentre(G); // compute centre of G  
  
> PCBFGrpOrder(Z) eq Order(Centre(E)); // orders should match  
  
true
```

**Example 6.3.** Computing a centraliser in a perfect group.

```
> D := PerfectGroupDatabase();  
  
> E := PermutationGroup(D, 120);  
  
> Order(E);  
  
375000  
  
> N := SolubleRadical(E);  
  
> Q, rho := RadicalQuotient(E);  
  
> G := PCBFMasterConstruct(E, N, Q, rho);  
  
> PCBFGrpOrder(G);  
  
375000  
  
> g := PCBFRandom(G);  
  
> C := PCBFCentraliser(G, g); // centraliser of g in G  
  
> PCBFGrpOrder(C);  
  
30
```

**Example 6.4.** Computing the derived group of a subgroup of  $\text{AGL}(4, 3)$ .

```
> n := 4; q := 3;

> G := AGL(n, q);

> N := MinimalNormalSubgroups(G)[1];

> M := MaximalSubgroups(Stabiliser(G, 1));

> gps := [ sub< G | N, m'subgroup > : m in M ];

> #gps;

8

> E := gps[1];

> Order(E);

233280

> S := SolubleRadical(E);

> Q, rho := RadicalQuotient(E);

> Order(Q);

1440

> H := PCBFMasterConstruct(E, S, Q, rho);

> L := PCBFDerivedGroup(H);

> PCBFGrpOrder(L) eq Order(DerivedGroup(E)); // should be true

true
```

**Example 6.5.** In this example, the `PCBFEleOrder` function is used to compute the order of an element in an automorphism group of a finite group.

```
> S := SmallGroup(729, 102);  
  
> A := AutomorphismGroup(S);  
  
> P := PermutationGroup(A);  
  
> Order(P);  
  
221079456  
  
> Degree(P);  
  
702  
  
> IsSoluble(P);  
  
false  
  
> N := SolubleRadical(P);  
  
> Q, rho := RadicalQuotient(P);  
  
> G := PCBFMasterConstruct(P, N, Q, rho);  
  
> PCBFGrpOrder(G);  
  
221079456  
  
> g := PCBFRandom(G);  
  
> PCBFEleOrder(G, g); // order of g  
  
24  
  
> H := PCBFsub(G, [g]); // subgroup generated by g  
  
> PCBFGrpOrder(H); // should be 24  
  
24
```



**Example 6.6.** Computing the centre of a reducible matrix group.

```
> D1 := ClassicalMaximals("L", 5, 2 : classes := 1);  
> D2 := ClassicalMaximals("L", 3, 2 : classes := 1);  
> X := [DirectProduct(d1, d2) : d1 in D1, d2 in D2];  
> IsIrreducible(X[1]);  
  
false  
  
> E := Image(PermutationRepresentation(X[1]));  
> Order(E);  
  
7741440  
  
> S := SolubleRadical(E);  
> Q, rho := RadicalQuotient(E);  
> Order(Q);  
  
20160  
  
> G := PCBFMasterConstruct(E, S, Q, rho);  
> Z := PCBFCentre(G);  
> PCBFGrpOrder(Z) eq Order(Centre(E)); // should be true  
  
true
```

**Example 6.7.** Let  $G$  be the projective special linear group acting on a vector space of dimension 5 over the field  $\mathbb{F}_2$ . The code that follows constructs an extension of an irreducible module of  $G$ , by  $G$ , as a record of type `PCBF`. The order of a random element of this extension is computed. See B.5 for details of constructions involving the type `ModCoho`.

```
> G := PSL(5, 2);  
  
> I := IrreducibleModules(G, GF(2));  
  
> C := CohomologyModule(G, I[2]); // module of dimension 5  
  
> H2 := CohomologyGroup(C, 2);  
  
> E, rho := Extension(C, H2.1);  
  
> EG := PCBFModCohoConstruct(C, E, rho);  
  
> y := PCBFRandom(EG);  
  
> PCBFEleOrder(EG, y);  
  
8
```

**Example 6.8.** In the code below, an extension of the natural permutation module over  $\mathbb{Z}$  of the alternating group,  $A_5$ , is constructed as a record of type `PCBF`. The derived subgroup of this extension is then computed. This is an example of an infinite polycyclic-by-finite group represented using the methods described in this thesis. See B.5 for details of constructions involving the type `ModCoho`.

```
> G := Alt(5);

> M := PermutationModule(G, Integers());

> C := CohomologyModule(G, M);

> _ := CohomologyGroup(C, 2);

> E, rho := Extension(C, [1]);

> EG := PCBFModCohoConstruct(C, E, rho);

> PCBFNormalSubgrp(EG);

GrpGPC of infinite order on 5 PC-generators

PC-Relations:

> X := PCBFDerivedGroup(EG);

> PCBFQuotGrp(X);

Permutation group acting on a set of cardinality 5

Order = 60 = 2^2 * 3 * 5

(1, 3, 4)

(2, 4)(3, 5)

(1, 4, 3)

(2, 3, 5)

(2, 5, 3)

(2, 5, 4)
```

**Example 6.9.** Consider the Heineken group defined by the finite presentation,

$$\langle a, b, c \mid [a, [a, b]] = c, [b, [b, c]] = a, [c, [c, a]] = b \rangle.$$

This group has a quotient with the structure  $[2^{24}] \cdot A_5$ . The snippet of code that follows constructs this quotient as a record of type PCBF and computes Sylow 2- and 3- subgroups. See B.5 for details in constructions involving quotients of finitely presented groups.

```
> E := Group< a, b, c | (a, (a, b)) = c, (b, (b, c)) = a, (c, (c, a)) = b>; //
Heineken group
> Q := Alt(5);
> rho := Homomorphisms(E, Q)[1];
> K := Kernel(rho);
> K := Rewrite(E, K : Simplify := false);
> L, psi := pQuotient(K, 2, 5 : Print := 1);
Lower exponent-2 central series for K
Group: K to lower exponent-2 central class 1 has order 2^5
Group: K to lower exponent-2 central class 2 has order 2^10
Group: K to lower exponent-2 central class 3 has order 2^16
Group: K to lower exponent-2 central class 4 has order 2^20
Group: K to lower exponent-2 central class 5 has order 2^24
> G := PCBFFPQuotConstruct(E, Q, rho, L, psi);
> Factorisation(PCBFGGrpOrder(G));
[<2, 26>, <3, 1>, <5, 1>]
> P2 := PCBFSylow(G, 2); // Compute Sylow 2-subgroup
> Factorisation(PCBFGGrpOrder(P2)); // should be 2^26
[<2, 26>]
> P3 := PCBFSylow(G, 3);
> Factorisation(PCBFGGrpOrder(P3)); // should be 3^1
[<3, 1>]
```

Tables 6.1–6.6 display a set of run-times in Magma for a variety of computations using the newly implemented suite of PCBF-group functions, over a collection of test groups. In particular, for each PCBF-group, run-times are measured for the following, each over 100 runs, choosing elements at random where applicable:

- (i) Multiplying a pair of elements
- (ii) Finding the inverse of an element
- (iii) In the finite case, finding a Sylow 2-subgroup and a Sylow  $p$ -subgroup, where  $p$  is the largest prime dividing the order of the group
- (iv) Computing the centre of the group

Tables 6.1–6.5 all follow the same format; the first and second columns of each table give the order (or in some cases, the structure) of the polycyclic normal subgroup,  $N$ , and the finite quotient,  $Q$ , respectively. The largest prime,  $p$ , dividing the order of group is recorded in the third column. The remaining columns give run-times as indicated, in seconds, and rounded to three decimal places. A “-” indicates that the table entry in question is not applicable for the particular example group, for instance, the run-time for a Sylow subgroup computation in an infinite group. In each table, the groups are ordered by increasing size of  $Q$ .

For each group listed in Tables 6.1–6.5, the first line of data records run-times using the PCBF representation of the group. For the sake of comparison, the second line of data gives run-times for similar computations using the existing representation of the group in Magma. In the case where there exists no native Magma method to perform the computation in question using the group’s existing representation, the time taken is indicated by “ $\infty$ ”. This occurs most often with finitely presented groups, for which no feasible permutation representation is available. There exist no intrinsic functions to compute the centre, or to find Sylow subgroups of a finitely

presented group in Magma. As a result, comparison run-times are unobtainable for such examples.

Table 6.1 gives run-times for a set of finite perfect groups. These groups are taken from the Magma Perfect Group Database, and their constructions follow the form of Examples 6.1–6.3.

Tables 6.2 and 6.3 investigate subgroups of  $\text{AGL}(3, 5)$  and  $\text{AGL}(4, 3)$  respectively. The standard Magma method is used to construct the groups,  $\text{AGL}(n, q)$ , and subgroups are generated as in Example 6.4.

Table 6.4 displays run-times for a collection of reducible matrix groups. These groups are constructed by forming direct products of the maximal subgroups of the classical groups as in Example 6.6. For a survey on the subgroup structure of the finite classical groups, the reader is referred to (Kleidman and Liebeck, 1990).

Table 6.5 explores the performance of the package for extensions of modules as described in Section 2.3. In each of the finite groups of Table 6.5, the extensions are created by first performing a search for all irreducible modules of the given matrix group, after which the Magma cohomology functions are utilised. Example 6.7 illustrates such a construction. The infinite examples of Table 6.5 are constructed as in Example 6.8, where a module of a suitable matrix group is constructed explicitly. All of the extensions of Table 6.5 are non-split, except for the fifth, for which only a split extension is possible.

Table 6.6 consists of a collection of test groups, each of which map onto  $A_5$ . For each test group, an appropriate  $p$ -quotient of the kernel of the epimorphism is computed, and then an extension of that quotient by  $A_5$  is constructed using the methods described in this thesis. See Example 6.9 for a sample construction.

The format of Table 6.6 is as follows. The first column gives the presentation of the large group, along with the map onto  $A_5$ . The second column gives data relating to the  $p$ -quotient, namely, its order and lower exponent- $p$  class. The third column records the largest prime,  $q$ , dividing the order of the extension. The remaining

columns give run-times as indicated, in seconds, and rounded to three decimal places. No comparison run-times are available for these test groups, as there is no existing standard method in Magma to construct extensions of this type.

It is important to note here, that the current implementations of the algorithms in this thesis are subject to the overhead of the Magma interpreter, which adds a significant time cost. As discussed in Subsection 6.2.3, it is expected that, after native implementation and optimisation, run-times will improve by a factor of roughly  $10^3$ . Tables of run-time data follow.

Table 6.1: Run-times for perfect groups

Group Information			Run-times (seconds)				
$N$	$Q$	$p$	$x \cdot y$	$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_p$	$\mathbf{Z}$
32	$A_5$	5	0.060	0.180	40.040	39.000	0.840
			0.000	0.000	0.000	0.010	0.000
3888	$A_5$	5	0.060	0.180	52.640	48.350	0.610
			0.000	0.000	0.000	0.000	0.000
6250	$A_5$	5	0.050	0.140	34.930	36.190	3.480
			0.000	0.000	0.000	0.010	0.000
7776	$A_5$	5	0.080	0.180	70.600	39.230	1.710
			0.000	0.000	0.000	0.000	0.000
15842	$A_5$	89	0.120	0.360	55.360	0.190	0.200
			0.000	0.000	0.000	0.010	0.000
128	$\text{PSL}(2, 7)$	7	0.120	0.330	99.160	94.840	1.470
			0.000	0.000	0.000	0.000	0.000
2187	$A_6$	5	0.210	0.470	156.660	80.630	9.370
			0.000	0.000	0.010	0.000	0.000
128	$A_7$	7	0.610	0.840	319.970	547.010	13.660
			0.000	0.000	0.000	0.000	0.000



Table 6.2: Run-times for subgroups of  $\text{AGL}(3, 5)$ 

Group Information			Run-times (seconds)				
$N$	$Q$	$p$	$x \cdot y$	$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_p$	$\mathbf{Z}$
46500	1	31	0.000	0.010	0.210	0.210	0.690
			0.000	0.000	0.000	0.000	0.000
48000	1	5	0.010	0.000	0.570	0.510	1.100
			0.000	0.000	0.000	0.000	0.000
500	120	5	0.100	0.310	44.660	50.510	0.290
			0.000	0.000	0.000	0.000	0.000
50000	120	5	0.100	0.240	100.150	59.450	1.340
			0.000	0.000	0.000	0.000	0.000
50000	120	5	0.160	0.380	95.140	85.510	1.020
			0.000	0.000	0.000	0.010	0.000
250	372000	31	2.810	3.090	4599.140	7793.520	0.190
			0.000	0.000	0.000	0.010	0.000

Table 6.3: Run-times for subgroups of  $\text{AGL}(4, 3)$ 

Group Information			Run-times (seconds)				
$N$	$Q$	$p$	$x \cdot y$	$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_p$	$\mathbf{Z}$
186624	1	3	0.000	0.010	1.250	1.200	1.820
			0.000	0.000	0.010	0.000	0.000
373248	1	3	0.000	0.010	1.800	1.680	2.350
			0.000	0.000	0.010	0.010	0.000
15116544	1	3	0.000	0.000	6.620	6.520	7.420
			0.000	0.000	0.010	0.010	0.000
162	1440	5	0.280	0.500	896.200	101.900	0.340
			0.000	0.000	0.000	0.000	0.000
648	1440	5	0.490	0.820	1013.610	142.860	0.500
			0.000	0.000	0.010	0.000	0.000
8748	5616	13	0.420	0.950	534.680	577.500	0.960
			0.000	0.000	0.010	0.010	0.000
8748	5616	13	0.760	1.250	546.410	905.560	0.940
			0.000	0.000	0.000	0.000	0.000

Table 6.4: Run-times for reducible matrix groups

Group Information			Run-times (seconds)				
$N$	$Q$	$p$	$x \cdot y$	$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_p$	$\mathbf{Z}$
13824	1	3	0.000	0.010	0.940	0.780	1.110
			0.000	0.000	0.000	0.000	0.000
192	PSL(2, 7)	7	0.140	0.360	96.260	125.460	5.100
			0.000	0.000	0.000	0.000	0.000
9216	PSL(2, 7)	7	0.170	0.400	97.770	91.610	1.230
			0.000	0.000	0.000	0.000	0.000
221184	PSL(2, 7)	7	0.210	0.480	143.750	67.260	2.300
			0.000	0.000	0.010	0.000	0.000
221184	PSL(2, 7)	7	0.240	0.500	102.510	124.070	1.800
			0.000	0.000	0.000	0.000	0.000
384	20160	7	0.950	1.120	429.910	432.050	12.250
			0.000	0.000	0.000	0.000	0.000

Table 6.5: Run-times for module extensions

Group Information			Run-times (seconds)				
$N$	$Q$	$p$	$x \cdot y$	$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_p$	$\mathbf{Z}$
$\mathbb{Z}^5$	$A_5$	-	0.070	0.180	-	-	2.900
			0.000	0.000	-	-	$\infty$
$\mathbb{Z}^7$	$\text{PSL}(2, 7)$	-	0.150	0.350	-	-	10.210
			0.000	0.000	-	-	$\infty$
$\mathbb{Z}^{10}$	$A_6$	-	0.290	0.410	-	-	13.750
			0.000	0.000	-	-	$\infty$
$(\mathbb{Z}/11\mathbb{Z})^3$	$\text{PSL}(2, 11)$	11	0.230	0.460	100.460	267.490	5.650
			0.000	0.000	$\infty$	$\infty$	$\infty$
$\mathbb{Z}^5$	960	-	0.800	1.450	-	-	169.930
			0.000	0.000	-	-	$\infty$
$\mathbb{Z}^7$	1344	-	0.570	1.020	-	-	107.550
			0.000	0.000	-	-	$\infty$
$(\mathbb{Z}/2\mathbb{Z})^4$	$\text{PSL}(4, 2)$	7	0.570	0.950	548.480	415.450	14.560
			0.000	0.000	$\infty$	$\infty$	$\infty$
$(\mathbb{Z}/2\mathbb{Z})^6$	$\text{PSU}(4, 2^2)$	5	0.610	1.100	2061.020	269.010	22.200
			0.000	0.000	$\infty$	$\infty$	$\infty$

Table 6.6: Run-times for finitely presented groups that map onto  $A_5$

Group Information			Run-times (seconds)					
Presentation and Map	$p$ -quotient of kernel	$q$	$x \cdot y$		$x^{-1}$	$\text{Syl}_2$	$\text{Syl}_q$	$\mathbf{Z}$
			Order	$p$ -class				
$\langle a, b, c \mid a^2 = c, b^3 = (ab)^5, cb^b = c^{ba} \rangle$ $a \mapsto (1\ 2)(3\ 4), b \mapsto (2\ 3\ 5), c \mapsto ()$	$2^5$	1	5	0.060	0.210	35.370	33.610	5.030
$\langle a, b, c \mid [a, [a, b]] = c, [b, [b, c]] = a, [c, [c, a]] = a \rangle$ $a \mapsto (1\ 2\ 3\ 4\ 5), b \mapsto (1\ 4\ 5\ 2\ 3), c \mapsto (1\ 4\ 2\ 3\ 5)$	$2^{24}$	5	5	0.150	0.270	98.840	47.360	12.910
$\langle a, b, c \mid a^2, b^3, (ab)^5, abcb^{-1}ac^{-1}b^{-1}c^{-1}bc^{-1} \rangle$ $a \mapsto (1\ 2)(3\ 4), b \mapsto (2\ 3\ 5), c \mapsto ()$	$13^{55}$	3	13	0.190	0.290	135.200	28.940	461.660
$\langle a, b, c \mid a^2 = (ab)^5, b^3, [a, b] = c^3, [b, c], [b, [b, c]] \rangle$ $a \mapsto (1\ 2)(3\ 4), b \mapsto (2\ 3\ 5), c \mapsto ()$	$3^{210}$	2	5	0.410	0.550	2772.530	2721.460	60793.650

The message to be taken from the data of Tables 6.1–6.6 is clear. Run-times increase significantly as the size of the finite quotient increases. This agrees with the complexity analysis of Theorem 3.5 and reinforces the content of the discussion immediately following that result. It should be noted that, the computation of Sylow-2 subgroups proved, in most cases, to be the most expensive computation, and that the run-time for computing the centre is dependent on the size of the normal polycyclic subgroup, as illustrated by the final example of Table 6.6.

One may conclude from the run-time data, that, with appropriate optimisations, the PCBF-group data structure is a suitable option for polycyclic-by-finite groups for which there is no available permutation or matrix representation, but whose finite quotient is small and easily represented as a permutation group.

## 6.2 Implementation

This section contains a description of the implementation of the polycyclic-by-finite group package using the Magma Computational Algebra System, highlighting a few technical details and efficiency considerations.

### 6.2.1 List of Functions

The author has implemented methods to perform the following operations in the class of polycyclic-by-finite groups:

- Multiply elements
- Compute the natural map (and its inverse) from a group to the finite quotient used in the construction
- Invert elements
- Compute the order of an element

- Transfer to category `GrpPC` or `GrpGPC`
- Find the subgroup generated by a set of elements
- Test element membership in a given group
- Coerce an element into a group
- Find the normal closure of the subgroup generated by a set of elements
- Compute the commutator subgroup of two groups having a common supergroup
- Compute Sylow subgroups (in the finite case)
- Compute the centre of a group
- Compute the derived series of a group
- Compute the lower central series of a group
- Compute the centraliser of an element
- Test whether two group elements are conjugate, and, if so, return a conjugating element

### 6.2.2 Technical Considerations

A major technical difficulty that arose during the course of the implementation was the inability to define Magma homomorphisms to and from polycyclic-by-finite groups represented using the newly defined structure.

The record (`Rec`) type in Magma was used to store the information defining a polycyclic-by-finite group, and since elements of type `Rec` are not recognised by Magma as algebraic structures (there is no binary operation), it was not possible to construct native homomorphisms to or from polycyclic-by-finite groups. This

problem was circumvented by maintaining data sets encoding homomorphisms, and computing images and preimages via helper methods.

The lack of a mechanism by which homomorphisms may be created proved especially problematic in the centraliser and element conjugacy functions, where, in the implementation, a finitely presented group is constructed and used to compute the maps of Line 11 in Algorithm 5.2 and Line 23 in Algorithm 5.3. At present, there is no feasible alternative to this costly solution. As a result, the implementations of the centraliser and element conjugacy functions, in their present state, should be regarded as experimental.

### 6.2.3 Native Support

In view of the author's progress in the area, Prof. John Cannon, Head of the Magma Computational Algebra Group, expressed interest in incorporating the methods developed by in to standard releases of the Magma Computational Algebra System, to enable built-in support of polycyclic-by-finite groups. This necessitated direct collaboration between the author and members of the Magma Computational Algebra Group.

To this end, in April, 2010, the author was invited by Prof. Cannon to visit the University of Sydney for a 5 week period. During this time, the author worked closely with Dr. William Unger to implement a set of functions for computing with polycyclic-by-finite groups.

The first stage of the implementation involved defining internal Magma data types for polycyclic-by-finite groups (New type: `GrpPCBF`) and for elements of polycyclic-by-finite groups (New type: `GrpPCBFelt`). These data types are defined in a similar fashion as for the current standard Magma group types, Permutation groups; `GrpPerm`, Matrix groups; `GrpMat`, Polycyclic groups; `GrpPC` (or `GrpGPC`), etc. The construction functions for these types mimic the construction functions of the



author's program code.

With these new types available, the author's package code was completely rewritten to correspond with the new type definitions.

Naturally, many of the methods implemented for computing with polycyclic-by-finite groups (listed in Section 6.2.1), depend heavily on element multiplication. As a result a significant portion of time was devoted to improving the performance of the multiplication algorithm.

After discussion with the author on the operation of the algorithm, a few optimisation adjustments were suggested by Dr. Unger. These yielded a noticeable improvement in efficiency. After this, to avoid the inherent overhead that is associated with the Magma interpreter, the entire multiplication algorithm was rewritten in the programming language C (the language in which the Magma Computational Algebra System is written) and then added to the standard Magma library. The implementation in C offered a significant performance gain — the speed of the function was improved by a factor of roughly  $10^3$ .

The results achieved were encouraging. The multiplication algorithm (and its dependants) performed well under most of the tests administered.

## 6.3 Future Work

There are several subareas of this project that are deserving of continued research. In addition to improvement in the performance of the multiplication algorithm, the following functionality is desirable:

- Find normalisers of subgroups
- Test whether two given subgroups are conjugate
- Find all conjugacy classes of a group
- Compute the character table of a group

Finding normalisers and testing subgroup conjugacy have immediate applications to computing within automorphism groups of  $p$ -groups. This forms part of ongoing research of Prof. Derek Holt and David Howden at the Warwick Mathematics Institute.

With regard to computing character tables of groups, the author plans to seek the expertise of Dr. William Unger in adapting his recently developed methods to the category of polycyclic-by-finite groups.

Another avenue which can be explored is the possibility of centraliser computation conjugacy testing in infinite polycyclic-by-finite groups. This may perhaps involve extending the work of Eick (2001).

The author is confident that, with continued efforts at optimisation, the suite of algorithms will soon prove to be even more useful from a practical point of view, in the contexts described here, and otherwise.

# Appendix A

## Package Documentation

### A.1 Introduction

The algorithms under development for polycyclic-by-finite groups employ current methods available for polycyclic presentations, together with the well developed base-and-strong-generating-set techniques used for computation with permutation groups. This document describes the data structure and the functions that have been implemented using the Magma Computational Algebra System. The current version at the time of writing is: Magma V2.17-5.

### A.2 Getting Started

To load all necessary function and record definitions, simply start Magma from the directory in which the source code files are held and execute the command:

```
load "pcbfmain.m";
```

Alternatively, to load all function and record definitions, and execute a series of tests on the source code, execute the command:

```
load "pcbfmain-test.m";
```

## A.3 Creation of a Group

To create a record of type `PCBF` representing a polycyclic-by-finite group the user is required to supply:

- (i) The polycyclic-by-finite group,  $E$ .
- (ii) A normal polycyclic subgroup,  $N$ , of  $E$  such that the associated quotient is finite.
- (iii) The quotient,  $E/N$ , (of type `GrpPerm` or `GrpMat`) and the natural map,  $E \rightarrow E/N$ .

### A.3.1 Construction Functions

`PCBFMasterConstruct(E, N, Q, rho)`

Constructs a record of type `PCBF` to represent the polycyclic-by-finite group,  $E$ , with normal polycyclic subgroup,  $N$ , and quotient group,  $Q$ .

In the method descriptions that follow,  $G$ ,  $H$  and  $K$  denote groups of this type.

## A.4 Basic Group Properties

The functions described here provide access to basic information stored for a `PCBF`-group,  $G$ .

### A.4.1 Infrastructure

`PCBFGenerators(G)`

An enumerated sequence containing a complete set of generators for  $G$ .

`PCBFNGens(G)`

`PCBFNumberOfGenerators(G)`

The number of defining generators for  $G$ .

`PCBFNormalSubgrp(G)`

A group of type `GrpPC` or `GrpGPC` (depending on whether  $G$  is finite or infinite respectively) isomorphic to the polycyclic normal subgroup used in the construction of the PCBF representation of  $G$ .

`PCBFQuotGrp(G)`

The quotient group used in the construction on the PCBF representation of  $G$ .

### A.4.2 Numerical Invariants

`PCBFGrpOrder(G)`

The order of  $G$ , returned as an ordinary integer.

### A.4.3 Predicates

`PCBFIsFinite(G)`

Returns `true` if  $G$  is finite, `false` otherwise.

`PCBFIsTrivial(G)`

Returns `true` if  $G$  has order 1, `false` otherwise.

`PCBFIsSoluble(G)`

`PCBFIsSolvable(G)`

Returns `true` if  $G$  is soluble, `false` otherwise.

## A.5 Elements

As discussed above, an element of a PCBF-group is stored as an ordered pair with components belonging to the quotient group and normal polycyclic subgroup respectively.

### A.5.1 Definition of Elements

`PCBFCreateElement(G, x, y)`

Given elements,  $x$ , of the quotient group, and,  $y$ , of the normal subgroup used in the construction of the PCBF representation of  $G$ , construct the element of  $G$  defined by the ordered pair,  $(x, y)$ .

`PCBFrho(G, g)`

Given an element,  $g$ , of  $G$ , return the image of  $g$  under the natural map onto the quotient group used in the construction of  $G$ .

`PCBFrhoinv(G, u)`

Given an element,  $u$ , of the quotient group used to construct  $G$ , return an inverse image of  $u$  in  $G$ .

`PCBFphiinv(G, n)`

Given an element,  $n$ , of the normal subgroup group used to construct  $G$ , return the image of  $n$  in  $G$  under the inclusion map.

`PCBFIdentity(G)`

`PCBFId(G)`

Construct the identity element of  $G$ .

### A.5.2 Arithmetic Operations on Elements

New elements can be computed from existing ones using the following implementations of standard functions.

`PCBFMult(G, g, h)`

`PCBFMultiply(G, g, h)`

`PCBFStar(G, g, h)`

The product of the elements,  $g$  and  $h$ , of  $G$ .

`PCBFInverse(G, g)`

`PCBFInv(G, g)`

The inverse of the element,  $g$ , of  $G$ .

`PCBFPower(G, g, n)`

The  $n$ -th power of the element,  $g$ , of  $G$ , where  $n$  is an integer.

`PCBFDivide(G, g, h)`

`PCBFSlash(G, g, h)`

The quotient of the element,  $g$ , of  $G$  by the element,  $h$ , of  $G$ , i.e. the element,  $g/h = gh^{-1}$ .

`PCBFConjugate(G, g, h)`

The conjugate of the element,  $g$ , of  $G$ , by the element  $h$ , of  $G$ , i.e. the element  $g^h = h^{-1}gh$ .

`PCBFCommutator(G, g, h)`

The commutator of the elements,  $g$  and  $h$ , of  $G$ , i.e. the element  $g^{-1}h^{-1}gh$ .

`PCBFSeqCommutator(G, [g1, ..., gn])`

Given the sequence,  $g_1, \dots, g_n$ , of elements of  $G$ , return the commutator  $(g_1, \dots, g_n)$ . Commutators are left-normed, so they are evaluated from left to right.

### A.5.3 Properties of Elements

`PCBFEleOrder(G, g)`

The order of the element,  $g$ , of  $G$ .

### A.5.4 Predicates for Elements

`PCBFeq(G, g, h)`

Given  $g$  and  $h$  belonging to  $G$  return `true` if  $g$  and  $h$  are the same element,

`false` otherwise.

`PCBFne(G, g, h)`

Given  $g$  and  $h$  belonging to  $G$  return `true` if  $g$  and  $h$  are distinct elements, `false` otherwise.

`PCBFIsIdentity(G, g)`

`PCBFIsId(G, g)`

Returns `true` if  $g$  is the identity element in  $G$  and `false` otherwise.

### A.5.5 Set Operations

`PCBFRandom(G)`

`PCBFRandomElement(G)`

Returns a pseudo-random element of  $G$ .

`PCBFRandomSequence(G, l)`

`PCBFRandomElementSequence(G, l)`

Given a non-negative integer,  $l$ , construct a sequence of length  $l$  of pseudo-random elements of  $G$ .

`PCBFRepresentative(G)`

`PCBFRep(G)`

A representative element of  $G$ . For a PCBF-group this always returns the identity element.

## A.6 Subgroups

Subgroups of PCBF-groups are treated as independent PCBF-groups in their own right, with the subgroup relationship maintained in internal data structures.



### A.6.1 Definition of Subgroups by Generators

`PCBFsub(G, L)`

Construct the subgroup of  $G$  generated by the elements of the sequence,  $L$ .

`PCBFnc1(G, L)`

Construct the normal closure of the subgroup of  $G$  generated by the elements of the sequence,  $L$ .

### A.6.2 Membership and Coercion

`PCBFin(G, g, H)`

Given an element,  $g$ , of  $G$  return **true** if  $g$  is an element of  $H$ , **false** otherwise.

`PCBFnotin(G, g, H)`

Given an element,  $g$ , of  $G$  return **true** if  $g$  is not an element of  $H$ , **false** otherwise.

`PCBFCoerce(G, g, H)`

Given an element,  $g$ , of  $G$  and another PCBF-group,  $H$ , attempt to write  $g$  as an element of  $H$ . If the rewrite is successful then the function returns **true** and  $g$  written as an element of  $H$ , otherwise the function returns **false** and an empty tuple.

### A.6.3 Standard Subgroup Constructions

`PCBFSubgroupConjugate(G, H, g)`

Construct the conjugate,  $g^{-1}Hg$ , of the group,  $H$ , under the action of the element,  $g$ , of  $G$ . The group,  $H$ , is assumed to be a subgroup of  $G$ .

`PCBFCommutatorSubgroup(H, K)`

Construct the commutator subgroup of groups,  $H$  and  $K$ , where  $H$  and  $K$  are subgroups of a common PCBF-group.

PCBFSubgroupNormalClosure( $G$ ,  $H$ )

The normal closure of the subgroup,  $H$ , in the group,  $G$ .

### A.6.4 Sylow Subgroups

PCBFSylowSubgroup( $G$ ,  $p$ )

PCBFSylow( $G$ ,  $p$ )

A Sylow  $p$ -subgroup for the finite PCBF-group,  $G$ .

PCBFSylowConjugatingElement( $G$ ,  $H$ ,  $K$ )

Given Sylow  $p$ -subgroups,  $H$  and  $K$ , of a finite PCBF-group,  $G$ , return the element of  $G$  conjugating  $H$  into  $K$ .

### A.6.5 Centralisers

PCBFCentraliser( $G$ ,  $g$ )

PCBFCentralizer( $G$ ,  $g$ )

The centraliser of the element,  $g$ , of the finite PCBF-group,  $G$ .

## A.7 Normal Subgroups and Subgroup Series

### A.7.1 Normal Structure

PCBFSolubleRadical( $G$ )

PCBFSolvableRadical( $G$ )

The soluble radical of  $G$ .

### A.7.2 Characteristic Subgroups

PCBFCentre( $G$ )

PCBFCenter( $G$ )

The centre of  $G$ .

`PCBFCommutatorSubgroup(G)`

`PCBFDerivedSubgroup(G)`

`PCBFDerivedGroup(G)`

The derived subgroup of  $G$ .

### A.7.3 Subgroup Series

`PCBFDerivedSeries(G)`

The derived series of  $G$ . The series is returned as a sequence of subgroups.

`PCBFLowerCentralSeries(G)`

The lower central series of  $G$ . The series is returned as a sequence of subgroups.

## A.8 Conjugacy

`PCBFIsConjugate(G, g, h)`

Given elements,  $g$  and  $h$ , belonging to a finite PCBF-group,  $G$ , return the value true if there exists  $c \in G$  such that  $g^c = h$ . If so, the function returns such a conjugating element as second value.

## A.9 Transfer Between Group Categories

### A.9.1 Transfer to GrpPC or GrpGPC

`PCBFPolycyclicGroup(G)`

Write  $G$  as a GrpPC or a GrpGPC if  $G$  is polycyclic. Attempts are made where possible to return a GrpPC. If the category transfer is successful, an ordered pair encoding the isomorphism is returned as a second value.

`PCBFPolycycliciso(G, P, isodata, g)`

Given a polycyclic group,  $P$ , isomorphic to  $G$ , return the image of the element,

$g$ , of  $G$  in  $P$  as defined by `isodata`.

`PCBFPolycyclicisoinv(G, isodata, h)`

Given an element,  $h$ , of a polycyclic group isomorphic to  $G$ , return the image of the element,  $h$ , in  $G$  as defined by `isodata`.

# Appendix B

## Program Listing

Listing B.1: grputils.m

---

```
1  /*  
  
   File :                               grputils.m  
  
6  Last modified :                     Fri , 07 Jan 2011 17:21:11 +0000  
  
   Author(s) :                        Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>  
11  
   Company :                           University of Warwick <http://www.warwick.ac.  
   uk>  
  
   Description :  
16  
   Miscellaneous group utilities .  
  
   Notes :  
21  
   MAGMA V2.16-13.  
  
   Copyright 2006-2010, University of Warwick. All rights reserved.  
26  
   */  
  
31 WordMultiply := function(posimgs , negimgs , word)
```

```

/*

Arguments:

36      posimgs: Sequence of group elements, Type(s): SetIndx,
          SeqEnum

          negimgs: Sequence of group elements, Type(s): SetIndx,
41      SeqEnum

          word: Sequence of integers representing a word in group
          elements indexed by posimgs and negimgs, Type(s): SeqEnum

46

Parameters:

Return Type(s):

51      GrpElt

Description:

56      Computes the product of the sequence of group elements
          represented by word.

*/

61      ans := Identity(Parent(posimgs[1]));
      for i in [1 .. #word] do
          ans *:= word[i] gt 0 select posimgs[word[i]] else negimgs[-
              word[i]];
      end for;
66      return ans;
end function;

```

---

## Listing B.2: permsv.m

```

/*

2      File:                permsv.m

Last modified:      Fri, 07 Jan 2011 17:21:11 +0000

7

Author(s):          Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

```

12 *Company:*                      *University of Warwick <http://www.warwick.ac.uk>*

*Description:*

17 *This file contains some auxiliary methods for working with  
orbits and Schreier  
vectors.*

*Notes:*

22 *MAGMA V2.16–13.*

*Copyright 2006–2010, University of Warwick. All rights reserved.*

27 *\*/*

*WordInverse := func < x | [ -y : y in Reverse(x) ] >; // inverse  
of a word*

32

*OrbitSV := function(alpha, X)*

*/\**

37

*Arguments:*

*alpha: Element of the set upon which the elements of X act,  
Type(s): Elt*

42

*X: Sequence of elements of a group acting on the set to  
which alpha belongs, Type(s): SeqEnum*

47 *Parameters:*

*Return Type(s):*

52 *SeqEnum, SeqEnum*

*Description:*

```

57      Computes the orbit of the point alpha under the action of
      X . The function returns the orbit and a Schreier
      vector
      corresponding to the orbit with indexing relative to the
      set, X.

62      */

      r := #X;
      v := [0];
      orb := [alpha];
67      orbsize := 1;
      c := 1;
      while c le orbsize do
        pt := orb[c];
        for i in [1 .. r] do
72          ppt := pt^X[i];
          if ppt notin orb then
            Append(~orb, ppt);
            orbsize += 1;
            v[orbsize] := i;
77          end if;
        end for;
        c += 1;
      end while;
      return v, orb;
82 end function;

```

BasicOrbitsSV := **function**(G, B, S)

```

87      /*

      Arguments:

      G: Permutation group, Type(s): GrpPerm
92      B: Base of G, Type(s): SeqEnum

      S: Strong generating set of G relative to B, Type(s):
      SetIndx
97

      Parameters:

102     Return Type(s):

```



*SeqEnum, SeqEnum*

107 *Description:*

*Computes the basic orbits and Schreier vectors (relative to B and S) of G.*

112 *\*/*

```

117  l := #B;
    if l eq 0 then return [], []; end if;
    m := #S;
    X := Isetseq(S);
    sv := [];
    bo := [];
    bs := [Stabiliser(G, B[1 .. i - 1]) : i in [1 .. l]];
    sv[1], bo[1] := OrbitSV(B[1], X);
122  for i in [2 .. l] do
    indexmap := [k : k in [1 .. m] | X[k] in bs[i]];
    sv[i], bo[i] := OrbitSV(B[i], X[indexmap]);
    for j in [2 .. #sv[i]] do
    sv[i][j] := indexmap[sv[i][j]];
127  end for;
    end for;
    return sv, bo;
end function;
```

132 SVWordInv := **function**(G, S, v, orb, pt)

*/\**

137 *Arguments:*

*G: Permutation group, Type(s): GrpPerm*

*S: Strong generating set for G, Type(s): SetIndx*

142

*v: Schreier vector corresponding to orbit orb (with indexing relative to S), Type(s): SeqEnum*

*orb: Sequence containing exactly the orbit of orb[1] under S, Type(s): SeqEnum*

147

*pt: Element of the set upon which G acts, Type(s): Elt*

152   *Parameters:*

*Return Type(s):*

157    *SeqEnum*

*Description:*

162    *Returns inverse of what SVWord should return, as an integer sequence i.e. a word in strong generators taking base point to pt as defined by the Schreier vector, v.*

*\*/*

```

167   pos := Position(orb, pt);
      r := v[pos];
      w := [];
      while r ne 0 do
172       Append(~w, -r);
         pt := r gt 0 select pt^(S[r]^(-1)) else pt^(S[-r]);
         pos := Position(orb, pt);
         r := v[pos];
      end while;
177   return WordInverse(w);
      end function;

```

SVPermutationInv := **function**(G, S, v, orb, pt)

182    */\**

*Arguments:*

187    *G: Permutation group, Type(s): GrpPerm*

*S: Strong generating set for G, Type(s): SetIndx*

*v: Schreier vector corresponding to orbit orb (with indexing relative to S), Type(s): SeqEnum*

*orb: Sequence containing exactly the orbit of orb[1] under S, Type(s): SeqEnum*

197    *pt: Element of the set upon which G acts, Type(s): Elt*

```

Parameters:

202 Return Type(s):

      GrpPermElt

207 Description:

      Returns inverse of what SVPermutation should return. i.e. a
      permutation (or matrix) at level of basic orbit orb taking
212 base point to pt.

      */

      pos := Position(orb, pt);
217 r := v[pos];
      u := Id(G);
      y := Id(G);
      while r ne 0 do
        y := r gt 0 select S[r]^-1 else S[-r];
222 u *:= y;
        pt := pt^y;
        pos := Position(orb, pt);
        r := v[pos];
      end while;
227 return u^-1;
end function;

```

---

## Listing B.3: recfrmtdef.m

---

```

/*
2
File:                                recfrmtdef.m

Last modified:                       Wed, 10 Nov 2010 16:37:18 +0000
7

Author(s):                           Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

12 Company:                          University of Warwick <http://www.warwick.ac.
    uk>

Description:

```

17 *The records used to represent PCBF-groups are defined here.*

*Notes:*

22 *MAGMA V2.16–13.*

*Copyright 2006–2010, University of Warwick. All rights reserved.*

27 *\*/*

*// define record formats to store the relevant data*

32

```
PDF := recformat<
  G: GrpPerm, // the permutation group itself
  natset: SetIndx, // the natural set on which G acts
  B: SeqEnum, // base of gp
37  bo: SeqEnum, // basic orbits of gp
  S: SetIndx, // set of strong generators of gp
  sgindexlist: SeqEnum, // sgindexlist[i] is list of indices of
    strong gens lying in the i-th basic stabiliser G[i]
  sv: SeqEnum, // sv[i] is Schreier vector for basic orbit
    number i
  tail: SeqEnum,
42  tailinv: SeqEnum,
  utail: SeqEnum,
  utailinv: SeqEnum
>;
```

47

```
PCBF := recformat<
  E, // master group, used for testing
  L, // normal subgroup, used for testing
  Q, // quotient E/L
  rho: Map, // natural homomorphism from E to Q, used for
    testing
52  GR: Rec, // record of type PDF
  strgenpreimgs: SetIndx, // preimages of strong generators
  strgeninvpreimgs: SetIndx, // preimages of inverses of strong
    generators
  N: Grp, // polycyclically presented group isomorphic to L
  phi: Map, // isomorphism from L onto N, used for testing
57  pcgens: SetIndx, // generators of polycyclic normal subgroup
  pcgenconjugates: SeqEnum,
  pcgenconjugatesinv: SeqEnum,
```

---

```

    tailelts: SeqEnum,
    taileltsinv: SeqEnum,
62  e: Tup, // identity element of this PCBF-group
    strgenpreimsgsf: SeqEnum,
    strgeninvpreimsgsf: SeqEnum,
    pcgensnf: SeqEnum, // sequence of elements (in normal form)
        that generate normal subgroup
    grpgens: SeqEnum, // sequence of elements (in normal form)
        that generate the entire group
67  ismaster: BoolElt, // true iff this record is created from a
    MAGMA-known group
    supergrp: Rec // parent group
>;

```

---



---

#### Listing B.4: permdata.m

---

```

1  /*

    File:                permdata.m

6  Last modified:        Fri, 07 Jan 2011 17:21:11 +0000

    Author(s):           Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

11  Company:             University of Warwick <http://www.warwick.ac.
    uk>

    Description:

16  The methods in this file are used to collect the required data
    from the
    finite quotient of a polycyclic-by-finite group.

21  Notes:

    MAGMA V2.16-13.

26  Copyright 2006-2010, University of Warwick. All rights reserved.

    */

31 load "grputils.m";

```

```

load "permsv.m";
load "recfrmtdef.m";

36 forward StrongGenWord, WordPermutation;

GrpPermData := function(G, B, S, sv, bo)

41 /*

Arguments:

G: Permutation group, Type(s): GrpPerm
46 B: Base of G, Type(s): SeqEnum

S: Strong generating set of G, Type(s): SetIndx

51 sv: Sequeunce of Schreier vectors relative to (B, S),
Type(s): SeqEnum

bo: Sequence of basic orbits relative to B, Type(s):
SeqEnum
56

Parameters:

61 Return Type(s):

Rec

66 Description:

This function collects the data required from a permutation
group that is to be subsequently viewed as a quotient of a
PCBF-group.

71 For some  $1 \leq i \leq \#B$ , some  $1 \leq j \leq \#bo[i]$ , let  $u$  be the
permutation defined by  $sv[i]$  which takes  $B[i]$  to  $bo[i][j]$ .
Let  $x_k$  be a strong generator belonging to  $S[i]$ , and let
 $h_1, h_2$  be the permutations defined by  $sv[i]$  which take
76  $B[i]$  to the image of  $B[i]$  under  $u * x_k, u * (x_k)^{-1}$ 
respectively.

Then

```

```

81       $u * x_k = t_1 * h_1$ 

      and

       $u * (x_k)^{-1} = t_2 * h_2$ 
86
      for some  $t_1, t_2$  in  $G[i + 1]$ . The elements  $tail[i][j][k]$ 
      and  $tailinv[i][j][k]$  are lists of integers defining words
      over  $S$ , which represent  $t_1$  and  $t_2$  respectively. If  $G$  is a
      quotient group,  $E/N$ , of some larger group,  $E$ , then
91
       $u * x_k = t_1 * h_1 * n_1$ 

      and

96       $u * (x_k)^{-1} = t_2 * h_2 * n_2$ 

      for some  $n_1, n_2$  in  $N$ . The entries  $utail[i][j][k]$  and
       $utailinv[i][j][k]$  act as pointers to the elements  $n_1$  and
       $n_2$ , a 0 value indicating that the normal subgroup element
101      is trivial.

      This function returns a record holding BSGS data for  $G$ 
      together with the arrays  $tail$ ,  $tailinv$ ,  $utail$  and  $utailinv$ 
      which are used in the construction of the PCBF-group of
106      which this group is a finite quotient.

      */

GR := rec<PDF | >;
GR`G := G;
111 GR`natset := GSet(G);
GR`B := B;
l := #GR`B;
GR`S := S;
116 m := #S;
GR`bo := bo;
GR`sv := sv;
bs := [Stabiliser(G, B[1 .. i - 1]) : i in [1 .. l]];
sgindexlist := [[1 .. m]];
121 for i in [2..l] do
    sgindexlist[i] := [k : k in [1 .. m] | S[k] in bs[i]];
end for;
GR`sgindexlist := sgindexlist;
tail := [];
126 tailinv := [];
utail := [];

```

```

    utoailinv := [];
    gct := 0;
    gctinv := 0;
131  for i in [1 .. l] do
        tail[i] := [];
        tailinv[i] := [];
        utoail[i] := [];
        utoailinv[i] := [];
136  orb := GR`bo[i];
        v := sv[i];
        sg := sgindexlist[i];
        orbsize := #orb;
        for j in [1 .. orbsize] do
141      tail[i][j] := [];
            tailinv[i][j] := [];
            utoail[i][j] := [];
            utoailinv[i][j] := [];
            pt := orb[j];
146      nsg := #sg;
            for k in [1 .. nsg] do
                ipt := pt^S[sg[k]];
                pos := Position(orb, ipt);
                if v[pos] eq sg[k] or v[j] eq -sg[k] then
151      // This image is a definition
                    tail[i][j][k] := [Integers() |];
                    utoail[i][j][k] := 0;
                else
                    tail[i][j][k] := StrongGenWord(GR, SVPermutationInv(G,
                        S, v, orb, pt)*S[sg[k]]*SVPermutationInv(G, S, v,
                        orb, ipt)^-1);
156      gct += 1;
                    utoail[i][j][k] := gct;
                end if;
                ipt := pt^(S[sg[k]]^-1);
                pos := Position(orb, ipt);
161      if v[pos] eq -sg[k] or v[j] eq sg[k] then
                    // This image is a definition
                    tailinv[i][j][k] := [Integers() |];
                    utoailinv[i][j][k] := 0;
                else
166      tailinv[i][j][k] := StrongGenWord(GR, SVPermutationInv
                    (G, S, v, orb, pt)*(S[sg[k]]^-1)*SVPermutationInv(G
                    , S, v, orb, ipt)^-1);
                    gctinv += 1;
                    utoailinv[i][j][k] := gctinv;
                end if;
            end for;
        end for;
171  end for;

```



```

    end for;
    GR`tail := tail;
    GR`utail := utail;
    GR`tailinv := tailinv;
176 GR`utailinv := utailinv;
    return GR;
end function;

181 StrongGenWord := function(GR, g)

    /*

    Arguments:

186     GR: Record of type PDF, Type(s): Rec

        g: Element of GR`G, Type(s): GrpPermElt

191     Parameters:

        Return Type(s):

196     SeqEnum

        Description:

201     Returns an integer sequence representing a word in the
        strong generators for g.

    */

206 w := [];
    l := #GR`B;
    for i in [1 .. l] do
        ipt := GR`B[i]^g;
211 w := SVWordInv(GR`G, GR`S, GR`sv[i], GR`bo[i], ipt) cat w;
        g := SVPermutationInv(GR`G, GR`S, GR`sv[i], GR`bo[i], ipt)
            ^-1;
    end for;
    return w;
end function;

216

StrongGenNormalForm := function(GR, g)

```

```

/*
221  Arguments:

      GR: Record of type PDF, Type(s): Rec

226  g: Element of GR`G, Type(s): GrpPermElt

      Parameters:

231  Return Type(s):

      SeqEnum

236  Description:

      Returns a 2-D integer sequence representing the normal form
      of g relative to the base and strong generating set held in
241 GR.

  */

  nf := [];
246  l := #GR`B;
  for i in [1 .. l] do
    ipt := GR`B[i]^g;
    nf[i] := SVWordInv(GR`G, GR`S ,GR`sv[i], GR`bo[i], ipt);
    g := WordPermutation(GR, nf[i])^-1;
251  end for;
  return nf;
end function;

256 WordPermutation := function(GR, w)

  /*

      Arguments:

261  GR: Record of type PDF, Type(s): Rec

      w: Integer sequence representing word over the strong
      generating set held in GR, Type(s): SeqEnum

266

```

*Parameters:*

271 *Return Type(s):*

*GrpPermElt*

276 *Description:*

*Returns the permutation of  $GR^G$  represented by the integer sequence  $w$ .*

281 *\*/*

```

g := Id(GR`G);
if IsEmpty(GR`S) then return g; end if;
wordlength := #w;
286 for i in [1 .. wordlength] do
    g *:= w[i] gt 0 select GR`S[w[i]] else (GR`S[-w[i]]) ^ -1;
end for;
return g;
end function;
```

---

#### Listing B.5: pcbfconstruct.m

---

*/\**

*File:* *pcbconstruct.m*

5

*Last modified:* *Fri, 07 Jan 2011 17:21:11 +0000*

*Author(s):* *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

10

*Company:* *University of Warwick <http://www.warwick.ac.uk>*

15 *Description:*

*The methods in this file enable the user to:*

1. *Construct records of type PCBF to represent polycyclic-by-finite group.*
2. *Access basic structural information about a polycyclic-by-finite group*

20 *represented in a record of type PCBF.*

*Notes:*

25 *MAGMA V2.16–13.*

*Copyright 2006–2010, University of Warwick. All rights reserved.*

30 *\*/*

**load** "permdata.m";

35

PCBFMasterConstruct := **function**(E, L, Q, rho)

*/\**

40 *Arguments:*

*E: Polycyclic-by-finite group, Type(s): Grp*

*L: Polycyclic normal subgroup of E, Type(s): GrpPerm,*  
45 *GrpMat*

*Q: Quotient E/L (finite), Type(s): GrpPerm*

*rho: Natural map E -> Q, Type(s): Map*

50

*Parameters:*

55 *Return Type(s):*

*Rec*

60 *Description:*

*Returns a record representing a PCBF-group isomorphic to E,*  
*with normal polycyclic subgroup and quotient isomorphic to L*  
*and Q respectively.*

65

*\*/*

```

EG := rec<PCBF | >;
EG`E := E;
70 EG`L := L;
EG`Q := Q;
B := Base(Q);
S := StrongGenerators(Q);
ReduceGenerators(~Q); // a nonredundant set of strong
    generators is required
75 S := StrongGenerators(Q);
bo := BasicOrbits(Q);
sv := SchreierVectors(Q);
GR := GrpPermData(Q, B, S, sv, bo);
EG`GR := GR;
80 EG`rho := rho;
strgenpreimgs := (GR`S)@@rho;
EG`strgenpreimgs := strgenpreimgs;
m := #strgenpreimgs;
strgeninvpreimgs := {@@}; // this set is needed in the case
    where a strong generator has order 2
85 for i in [1 .. m] do
    Include(~strgeninvpreimgs, (Order(S[i]) eq 2) select
        strgenpreimgs[i] else strgenpreimgs[i]^(-1));
end for;
EG`strgeninvpreimgs := strgeninvpreimgs;
// the following only works if L is of the type GrpPerm,
    GrpMat, GrpAb or GrpGPC, phi : L -> N is an isomorphism
    onto the polycyclically presented group, N
90 if IsFinite(L) then
    N, phi := PCGroup(L);
else
    N, phi := GPCGroup(L);
end if;
95 pcgens := PCGenerators(N);
r := #pcgens;
pcgenconjugates := [];
pcgenconjugatesinv := [];
for i in [1 .. m] do
100 pcgenconjugates[i] := [N | ];
pcgenconjugatesinv[i] := [N | ];
for j in [1 .. r] do
    // computing conjugates of the j-th polycyclic generator
        with the ith preimage of the strong generating set of Q
        and its inverse
    yim := (pcgens[j])@@phi;
105 pcgenconjugates[i][j] := phi(yim^strgenpreimgs[i]); //
        stored as an element of N
pcgenconjugatesinv[i][j] := phi(yim^strgeninvpreimgs[i]);
    // stored as an element of N

```

```

    end for;
end for;
l := #GR`B; // number of elements in the stored base of the
           quotient group
110 tail := GR`tail;
    tailinv := GR`tailinv;
    utail := GR`utail;
    utailinv := GR`utailinv;
h := [];
115 g := [];
    tailelts := [N | ];
    taileltsinv := [N | ];
    for i in [1 .. l] do
        orb := GR`bo[i];
120 v := GR`sv[i];
        sg := GR`sgindexlist[i];
        for j in [1 .. #orb] do
            pt := orb[j];
            for k in [1 .. #sg] do
125 g := SVWordInv(Q, S, v, orb, pt);
                if utail[i][j][k] ne 0 then
                    ipt := pt^S[sg[k]];
                    h := SVWordInv(Q, S, v, orb, ipt);
                    ux := WordMultiply(strgenpreimgs, strgeninvpreimgs, g)
                        * strgenpreimgs[sg[k]];
130 th := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                        tail[i][j][k]) * WordMultiply(strgenpreimgs,
                        strgeninvpreimgs, h);
                    tailelts[utail[i][j][k]] := phi((th^-1) * ux);
                end if;
                if utailinv[i][j][k] ne 0 then
                    ipt := pt^(S[sg[k]]^-1);
135 h := SVWordInv(Q, S, v, orb, ipt);
                    uxinv := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                    g) * strgeninvpreimgs[sg[k]];
                    th := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                    tailinv[i][j][k]) * WordMultiply(strgenpreimgs,
                    strgeninvpreimgs, h);
                    taileltsinv[utailinv[i][j][k]] := phi((th^-1) * uxinv)
                    ;
                end if;
            end for;
140 end for;
        end for;
    end for;
EG`N := N;
EG`pcgens := pcgens;
145 EG`pcgenconjugates := pcgenconjugates;
EG`pcgenconjugatesinv := pcgenconjugatesinv;

```

```

EG`tailelts := tailelts;
EG`taileltsinv := taileltsinv;
e := <StrongGenNormalForm(GR, Id(GR`G)), Id(N)>;
150 EG`e := e;
EG`ismaster := true;
EG`strgenpreimsgsnf := [];
EG`strgeninvpreimsgsnf := [];
for i in [1 .. #GR`S] do
155   EG`strgenpreimsgsnf[i] := <StrongGenNormalForm(GR, S[i]), Id(
      N)>;
   EG`strgeninvpreimsgsnf[i] := <StrongGenNormalForm(GR, S[i
      ]^-1), Id(N)>;
end for;
EG`pcgensnf := [];
for i in [1 .. #pcgens] do
160   EG`pcgensnf[i] := <e[1], pcgens[i]>;
end for;
EG`phi := phi;
EG`grpgens := EG`strgenpreimsgsnf cat EG`pcgensnf;
EG`supergrp := rec<PCBF | >;
165 return EG;
end function;

```

```

PCBFPrecomputedDataConstruct := function(GR, N, pcgens,
    pcgenconjugates, pcgenconjugatesinv, tailelts, taileltsinv)
170
    /*

```

*Arguments:*

```

175   GR: Record containing permutation group data, Type(s): Rec

   N: Normal Subgroup, Type(s): GrpPC or GrpGPC

   pcgens: Generating sequence of N, Type(s): SeqEnum
180
   pcgenconjugates: 2-D sequence of conjugates, Type(s):
      SeqEnum

   pcgenconjugatesinv: 2-D sequence of conjugates, Type(s):
185   SeqEnum

   tailelts: Sequence of tail elements, Type(s): SeqEnum

   taileltsinv: Sequence of tail elements, Type(s): SeqEnum
190

```

*Parameters:*

195 *Return Type(s):*

*Rec*

200 *Description:*

*This function constructs a record representing a PCBF-group from the data provided. The exact specifications of the arguments are given below.*

205

*The record, GR, is assumed to be output from the function GrpPermData. See the documentation in the function header for details.*

210

*The group, N, is the normal polycyclic subgroup of the polycyclic-by-finite group in question, of type either GrpPC or GrpGPC.*

215

*The enumerated sequence pcgens contains the generating sequence of N used to compute pcgenconjugates and pcgenconjugatesinv.*

220

*Let S be the strong generating set held in the record GR. The 2-D sequence pcgenconjugates contains the image of each element of pcgens under the conjugation action of each element of S. Specifically,*

$$pcgenconjugates[i][j] = pcgens[j]^{S[i]}.$$

225

*Similarly,*

$$pcgenconjugatesinv[i][j] = pcgens[j]^{(S[i]^{-1})}.$$

230

*The enumerated sequence tailelts contains the elements of N that result when one attempts to right multiply (in the proposed polycyclic-by-finite group) a basic transversal element of the base and strong generating structure held in GR by a strong generator in S.*

235

*Let G be the permutation group held in GR, B its base and assume that B defines the stabiliser chain*

$$G[1] > G[2] > \dots > G[m+1] = 1.$$



```

240      For  $1 \leq i \leq \#B$ ,  $1 \leq j \leq \#GR\text{'}bo[i]$ , let  $g$  be the
      permutation defined by  $sv[i]$  which takes  $B[i]$  to  $bo[i][j]$ .
      Let  $x\_k$  be a strong generator belonging to  $S[i]$ , and let  $h$ 
      be the permutation defined by  $sv[i]$  which take  $B[i]$  to the
      image of  $B[i]$  under  $g * x\_k$ .

245      Then

      
$$g * x\_k = t * h * n$$


250      for some  $t$  in  $G[i + 1]$ ,  $n$  in  $N$ . The element,  $t$ , is encoded
      by the integer list  $GR\text{'}tail[i][j][k]$  as described in the
      function header of GrpPermData. The element,  $n$ , is stored
      in tailelts as follows:

255      
$$tailelts[GR\text{'}utail[i][j][k]] = n,$$


      where  $GR\text{'}utail$  is the sequence of integer pointers computed
      in GrpPermData. Likewise,

260      
$$taileltsinv[GR\text{'}utailinv[i][j][k]] = n',$$


      where

      
$$g * (x\_k)^{-1} = t' * h' * n'$$


265      for some  $t'$  in  $G[i + 1]$  (encoded by  $tailinv[i][j][k]$ ),  $n'$ 
      in  $N$ .

      A record representing a PCBF-group is returned.

270      Given the complicated nature of the arguments to this
      function, the easiest course to follow may be to copy and
      modify the code in the function PCBFMasterConstruct to suit
      the particular example with which you wish to work.

275      */

      EG := rec<PCBF | >;
      EG`Q := GR`G;
280      S := GR`S;
      EG`GR := GR;
      EG`N := N;
      EG`pcgens := pcgens;
      EG`pcgenconjugates := pcgenconjugates;
285      EG`pcgenconjugatesinv := pcgenconjugatesinv;
      EG`tailelts := tailelts;
      EG`taileltsinv := taileltsinv;

```

```

    e := <StrongGenNormalForm(GR, Id(GR`G)), Id(N)>;
    EG`e := e;
290 EG`ismaster := true;
    EG`strgenpreimgsnf := [];
    EG`strgeninvpreimgsnf := [];
    for i in [1 .. #GR`S] do
        EG`strgenpreimgsnf[i] := <StrongGenNormalForm(GR, S[i]), Id(
            N)>;
295 EG`strgeninvpreimgsnf[i] := <StrongGenNormalForm(GR, S[i
            ]^-1), Id(N)>;
    end for;
    EG`pcgensnf := [];
    for i in [1 .. #pcgens] do
        EG`pcgensnf[i] := <e[1], pcgens[i]>;
300 end for;
    EG`grpgens := EG`strgenpreimgsnf cat EG`pcgensnf;
    EG`supergrp := rec<PCBF | >;
    return EG;
end function;
305

PCBFModCohoConstruct := function(C, E, rho)

    /*
310 Arguments:

        C: Cohomology Module, Type(s): ModCoho

315 E, rho : Result of call: E, rho := Extension(C, ?), Type(s):
        Grp, Hom

        Return Type(s):

320 Rec

        Description:

325 Returns a record representing a PCBF-group isomorphic to E,
        with normal abelian subgroup Module(C) and quotient
        isomorphic to Group(C).

330 */

    EG := rec<PCBF | >;
        EG`E := E;

```

```

        Q := Group(C);
335 EG`Q := Group(C);
    B := C`gr`b;
    S := C`gr`sg;
    bo := C`gr`bo;
    sv := C`gr`sv;
340 GR := GrpPermData(Q, B, S, sv, bo);
    EG`GR := GR;
        EG`rho := rho;
    M := Module(C);
    p := IsFinite(BaseRing(M)) select #BaseRing(M) else 0;
345 r := Dimension(M);
    assert Ngens(E) eq #S + r;
    L := sub<E | [E.i : i in [#S + 1 .. Ngens(E)]]>;
    EG`L := L;
    N := p ne 0 select AbelianGroup(GrpPC, [p : i in [1 .. r
        ]])
350 else AbelianGroup(GrpGPC, [p : i in [1 .. r]]);
    EG`N := N;
    phi := hom<L -> N | [N.i : i in [1 .. r]]>;
    strgenpreimgs := (GR`S)@@rho;
    m := #strgenpreimgs;
355 EG`strgenpreimgs := strgenpreimgs;
    strgeninvpreimgs := {@@}; // this set is needed in the
        case where a strong generator has order 2
    for i in [1 .. m] do
        Include(~strgeninvpreimgs, (Order(S[i]) eq 2)
            select strgenpreimgs[i] else strgenpreimgs[i]
                ^-1);
    end for;
360 EG`strgeninvpreimgs := strgeninvpreimgs;
    pcgens := PCGenerators(N);
    EG`pcgens := pcgens;
    pcgenconjugates := [];
    pcgenconjugatesinv := [];
365 NF := E`NormalForm;
    for i in [1 .. m] do
        pcgenconjugates[i] := [N | ];
        pcgenconjugatesinv[i] := [N | ];
        for j in [1 .. r] do
370 // computing conjugates of the jth polycyclic generator
        with the ith preimage of the strong generating set of Q
        and its inverse
        yim := (pcgens[j])@@phi;
        pcgenconjugates[i][j] := phi(NF(yim^strgenpreimgs[i])); //
            stored as an element of N
            pcgenconjugatesinv[i][j] := phi(NF(yim^
                strgeninvpreimgs[i])); // stored as

```

*an element of  $N$*

```

    end for;
375 end for;
    l := #GR`B; // number of elements in the stored base of the
           quotient group
    tail := GR`tail;
    tailinv := GR`tailinv;
    utail := GR`utail;
380 utailinv := GR`utailinv;
    h := [];
    g := [];
    w := [];
    tailelts := [N | ];
385 taileltsinv := [N | ];
    for i in [1 .. l] do
        orb := GR`bo[i];
        v := GR`sv[i];
        sg := GR`sgindexlist[i];
390 for j in [1 .. #orb] do
            pt := orb[j];
            for k in [1..#sg] do
                g := SVWordInv(Q, S, v, orb, pt);
                if utail[i][j][k] ne 0 then
395 ipt := pt^S[sg[k]];
                    h := SVWordInv(Q, S, v, orb, ipt);
                    h := SVWordInv(Q, S, v,
                                   orb, ipt);
                    ux := WordMultiply(
                        strgenpreimgs,
                        strgeninvpreimgs, g)
                        * strgenpreimgs[sg[k]
                        ];
                    th := WordMultiply(
                        strgenpreimgs,
                        strgeninvpreimgs,
                        tail[i][j][k]) *
                        WordMultiply(
                            strgenpreimgs,
                            strgeninvpreimgs, h);
400 tailelts[utail[i][j][k]]
                        := phi(NF(th^-1 * ux
                        ));
            end if;
            if utailinv[i][j][k] ne 0 then
                ipt := pt^(S[sg[k]]^-1);
                h := SVWordInv(Q, S, v, orb, ipt);
405 uxinv := WordMultiply(strgenpreimgs, strgeninvpreimgs, g) *
                strgeninvpreimgs[sg[k]];

```

```

th := WordMultiply(
    strgenpreimgs ,
    strgeninvpreimgs ,
    tailinv[i][j][k]) *
    WordMultiply(
    strgenpreimgs ,
    strgeninvpreimgs , h);
taileltsinv[utailinv[i][
j][k]] := phi(NF(th
^-1 * uxinv));

    end if;
    end for;
410  end for;
    end for;
    EG`pcgenconjugates := pcgenconjugates;
    EG`pcgenconjugatesinv := pcgenconjugatesinv;
    EG`tailelts := tailelts;
415  EG`taileltsinv := taileltsinv;
    e := <StrongGenNormalForm(GR, Id(GR`G)), Id(N)>;
    EG`e := e;
    EG`ismaster := true;
    EG`strgenpreimgsnf := [];
420  EG`strgeninvpreimgsnf := [];
    for i in [1 .. #GR`S] do
        EG`strgenpreimgsnf[i] := <StrongGenNormalForm(GR, S[i]), Id(
            N)>;
        EG`strgeninvpreimgsnf[i] := <StrongGenNormalForm
            (GR, S[i]^ -1), Id(N)>;
    end for;
425  EG`pcgensnf := [];
    for i in [1 .. #pcgens] do
        EG`pcgensnf[i] := <e[1], pcgens[i]>;
    end for;
    EG`phi := phi;
430  EG`grpgens := EG`strgenpreimgsnf cat EG`pcgensnf;
    EG`supergrp := rec<PCBF | >;
    return EG;
end function;

435  PCBFFPQuotConstruct := function(G, Q, rho, L, psi)

    /*

440  Arguments:

        G: Finitely presented group, Type(s): GrpFP

```

```

    Q: Quotient of E via rho, Type(s): GrpPerm
445
    rho: Natural map E -> Q, Type(s): Map

    L: Quotient of ker(rho) via psi Type(s): GrpPC, GrpGPC
450
    psi: Natural map ker(rho) -> L, Type(s): Map

    Parameters:

455
    Return Type(s):

    Rec

460
    Description:

    Returns a record representing a PCBF-group isomorphic to
    the extension of L by Q.

465
    */

    N := L;
    // psi : ker(rho) -> L
470
    EG := rec<PCBF | >;
    EG`E := G;
    EG`L := L;
    EG`Q := Q;
    B := Base(Q);
475
    S := StrongGenerators(Q);
    ReduceGenerators(~Q); // a nonredundant set of strong
        generators is required
    S := StrongGenerators(Q);
    bo := BasicOrbits(Q);
    sv := SchreierVectors(Q);
480
    GR := GrpPermData(Q, B, S, sv, bo);
    EG`GR := GR;
    strgenpreimgs := (GR`S)@@rho;
    EG`strgenpreimgs := strgenpreimgs;
    m := #strgenpreimgs;
485
    strgeninvpreimgs := {@@}; // this set is needed in the case
        where a strong generator has order 2
    for i in [1 .. m] do
        Include(~strgeninvpreimgs, (Order(S[i]) eq 2) select
            strgenpreimgs[i] else strgenpreimgs[i]^(-1));
    end for;

```

```

EG`strgeninvpreimgs := strgeninvpreimgs;
490 pcgens := PCGenerators(N);
    r := #pcgens;
    pcgenconjugates := [];
    pcgenconjugatesinv := [];
    for i in [1 .. m] do
495     pcgenconjugates[i] := [N | ];
        pcgenconjugatesinv[i] := [N | ];
        for j in [1 .. r] do
            // computing conjugates of the j-th polycyclic generator
            // with the ith preimage of the strong generating set of Q
            // and its inverse
            yim := (pcgens[j])@@psi;
500     pcgenconjugates[i][j] := psi(yim^strgenpreimgs[i]); //
            // stored as an element of N
            pcgenconjugatesinv[i][j] := psi(yim^strgeninvpreimgs[i]);
            // stored as an element of N
        end for;
    end for;
    l := #GR`B; // number of elements in the stored base of the
            // quotient group
505    tail := GR`tail;
    tailinv := GR`tailinv;
    utail := GR`utail;
    utailinv := GR`utailinv;
    h := [];
510    g := [];
    tailelts := [N | ];
    taileltsinv := [N | ];
    for i in [1 .. l] do
        orb := GR`bo[i];
515        v := GR`sv[i];
        sg := GR`sgindexlist[i];
        for j in [1 .. #orb] do
            pt := orb[j];
            for k in [1 .. #sg] do
520                g := SVWordInv(Q, S, v, orb, pt);
                if utail[i][j][k] ne 0 then
                    ipt := pt^S[sg[k]];
                    h := SVWordInv(Q, S, v, orb, ipt);
                    ux := WordMultiply(strgenpreimgs, strgeninvpreimgs, g)
                        * strgenpreimgs[sg[k]];
525                th := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                    tail[i][j][k]) * WordMultiply(strgenpreimgs,
                    strgeninvpreimgs, h);
                    tailelts[utail[i][j][k]] := psi((th^-1) * ux);
                end if;
                if utailinv[i][j][k] ne 0 then

```

```

                    ipt := pt^(S[sg[k]]^ -1);
530      h := SVWordInv(Q, S, v, orb, ipt);
      uxinv := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                           g) * strgeninvpreimgs[sg[k]];
      th := WordMultiply(strgenpreimgs, strgeninvpreimgs,
                        tailinv[i][j][k]) * WordMultiply(strgenpreimgs,
                                                           strgeninvpreimgs, h);
      taileltsinv[utailinv[i][j][k]] := psi((th^ -1) * uxinv)
      ;
      end if;
535   end for;
      end for;
      end for;
      EG`N := N;
      EG`pcgens := pcgens;
540   EG`pcgenconjugates := pcgenconjugates;
      EG`pcgenconjugatesinv := pcgenconjugatesinv;
      EG`tailelts := tailelts;
      EG`taileltsinv := taileltsinv;
      e := <StrongGenNormalForm(GR, Id(GR`G)), Id(N)>;
545   EG`e := e;
      EG`ismaster := true;
      EG`strgenpreimgsnf := [];
      EG`strgeninvpreimgsnf := [];
      for i in [1 .. #GR`S] do
550   EG`strgenpreimgsnf[i] := <StrongGenNormalForm(GR, S[i]), Id(
      N)>;
      EG`strgeninvpreimgsnf[i] := <StrongGenNormalForm(GR, S[i
      ]^ -1), Id(N)>;
      end for;
      EG`pcgensnf := [];
      for i in [1 .. #pcgens] do
555   EG`pcgensnf[i] := <e[1], pcgens[i]>;
      end for;
      EG`phi := IdentityHomomorphism(L);
      EG`grpgens := EG`strgenpreimgsnf cat EG`pcgensnf;
      EG`supergrp := rec<PCBF | >;
560   return EG;
end function;

```

PCBFGenerators := function(EG)

565

/\*

*Arguments:*

570

EG: PCBF-group, Type(s): Rec



```

Parameters:

575 Return Type(s):

      SeqEnum

580 Description:

      Returns a sequence containing a full generating set for EG.

585 */

      return EG`grpgens;
end function;

590 PCBFNgens := function (EG)

      /*

595 Arguments:

      EG: PCBF-group, Type(s): Rec

      Parameters:

600 Return Type(s):

      RngIntElt

605 Description:

      Returns the number of defining generators for EG.

610 */

      return #EG`grpgens;
end function;

615 PCBFNumberOfGenerators := function (EG)

```

---

```

        /*
620    Arguments:

        EG: PCBF-group, Type(s): Rec

625    Parameters:

        Return Type(s):

630    RngIntElt

        Description:

635    Returns the number of defining generators for EG.

        */

    return PCBFNgens(EG);
640 end function;

PCBFQuotGrp := function(EG)

645    /*

    Arguments:

        EG: PCBF-group, Type(s): Rec

650

    Parameters:

655    Return Type(s):

        GrpPerm

660    Description:

        Returns a permutation group isomorphic to the quotient group
        used to construct EG.

665    */

```

```

    return EG`Q;
end function;

```

670

```
PCBFNormalSubgrp := function(EG)

```

```
    /*

```

675

```
    Arguments:

```

```
        EG: PCBF-group, Type(s): Rec

```

680

```
    Parameters:

```

```
        Return Type(s):

```

685

```
        GrpPC or GrpGPC

```

```
    Description:

```

690

```
        Returns a polycyclically presented group isomorphic to the
        normal subgroup group used to construct EG.

```

```
    */

```

695

```
    return EG`N;
end function;

```

```
PCBFSuperGrp := function(EG)

```

700

```
    /*

```

```
    Arguments:

```

705

```
        EG: PCBF-group, Type(s): Rec

```

```
    Parameters:

```

710

```
        Return Type(s):

```

```
        Rec

```

715

*Description :**Returns the PCBF-group from which EG was primarily derived.*

720

\*/

```

return EG`ismaster select EG else EG`supergrp;
end function;

```

725

PCBFGrpOrder := **function**(EG)

/\*

730

*Arguments :**EG: PCBF-group, Type(s): Rec*

735

*Parameters :**Return Type(s):*

740

*RngIntElt**Description :*

745

*Returns the order of the PCBF-group EG.*

\*/

```

return Order(PCBFQuotGrp(EG)) * Order(PCBFNormalSubgrp(EG));
750 end function;

```

PCBFIsFinite := **function**(EG)

755

/\*

*Arguments :**EG: PCBF-group, Type(s): Rec*

760

*Parameters :*

```

765  Return Type(s):

      BoolElt

770  Description:

      Returns true if and only if EG is finite.

      */
775  return IsFinite(PCBFGrpOrder(EG));
end function;

780 PCBFIstTrivial := function(EG)

      /*

      Arguments:

785   EG: PCBF-group, Type(s): Rec

      Parameters:

790

      Return Type(s):

      BoolElt

795

      Description:

      Returns true if and only if EG is trivial.

800  */

      return PCBFGrpOrder(EG) eq 1;
end function;

805

PCBFIstSoluble := function(EG)

      /*

810

```

```

      Arguments :

      EG: PCBF-group , Type(s): Rec

815  Parameters :

      Return Type(s):

820  BoolElt

      Description :

825  Returns true if and only if EG is soluble .

      */

830  return IsSoluble (PCBFQuotGrp(EG)) ;
end function ;

PCBFIsSolvable := function (EG)

835  /*

      Arguments :

840  EG: PCBF-group , Type(s): Rec

      Parameters :

845  Return Type(s):

      BoolElt

850  Description :

      Returns true if and only if EG is soluble .

855  */

      return PCBFIsSoluble (EG) ;
end function ;

```

```

860 PCBFCreateElement := function(EG, g, n)

    /*

865   Arguments :

        EG: PCBF-group, Type(s): Rec

        g: Element of EG`Q, Type(s): GrpPermElt

870        n: Element of EG`N, Type(s): GrpPCElt or GrpGPCElt

        Parameters :

875

        Return Type(s):

            Tup

880

        Description :

            Returns an ordered pair representing the element, (g, n),
885            of EG.

            */

    GR := EG`GR;
890    return <StrongGenNormalForm(GR, g), n>;
end function;

PCBFIdentity := function(EG)

895    /*

        Arguments :

900        EG: PCBF-group, Type(s): Rec

        Parameters :

905

        Return Type(s):

```

```

    Tup

910    Description :

        Returns the identity element of EG.

915    */

    return EG`e;
end function;

920    PCBFId := function(EG)

        /*

925    Arguments :

        EG: PCBF-group, Type(s): Rec

930    Parameters :

        Return Type(s):

935    Tup

        Description :

940    Returns the identity element of EG.

        */

    return PCBFIdentity(EG);
945 end function;

    PCBFrho := function(EG, ele)

950    /*

        Arguments :

        EG: PCBF-group, Type(s): Rec

```



955

*ele*: *Element of EG*, *Type(s)*: *Tup*

*Parameters*:

960

*Return Type(s)*:

*GrpPermElt*

965

*Description*:

970 *Returns the image of the element ele of EG under the  
natural map onto the quotient EG`Q.*

*\*/*

word := [];  
975 **for** i **in** [#ele[1] .. 1 **by** -1] **do**  
    word **cat**:= ele[1][i]; *// first entry specifies a word in the*  
        *strong generators of the quotient group*  
**end for**;  
    **return** WordPermutation(EG`GR, word);  
**end function**;

980

PCBFphi := **function**(EG, ele)

*/\**

985

*Arguments*:

*EG*: *PCBF-group*, *Type(s)*: *Rec*

990

*ele*: *Element of EG*, *Type(s)*: *Tup*

*Parameters*:

995

*Return Type(s)*:

*GrpPCElt or GrpGPCElt*

1000

*Description*:

```

    Returns the second component of the ordered pair, ele. This
    is NOT a homomorphism as no check is made to determine
1005    whether ele represents an element of the normal subgroup
    used in the construction of EG.

    */

1010    return ele[2];
    end function;

PCBFphiinv := function(EG, n)
1015    /*

    Arguments:

1020    EG: PCBF-group, Type(s): Rec

        ele: Element of EG`N, Type(s): GrpPCElt or GrpGPCElt

1025    Parameters:

        Return Type(s):

1030    Tup

    Description:

1035    Returns image in EG of the element, n, of EG`N.

    */

    return <PCBFId(EG)[1], n>;
1040 end function;

PCBFeq := function(EG, ele1, ele2)

1045    /*

    Arguments:

        EG: PCBF-group, Type(s): Rec

```

```

1050      ele1: Element of EG, Type(s): Tup

      ele2: Element of EG, Type(s): Tup

1055      Parameters:

      Return Type(s):

1060      BoolElt

      Description:

1065      Returns true if and only if ele1 and ele2 are equal in EG.

      */

1070      return PCBFrho(EG, ele1) eq PCBFrho(EG, ele2) and PCBFphi(EG,
      ele1) eq PCBFphi(EG, ele2);
end function;

PCBFne := function(EG, ele1, ele2)
1075      /*

      Arguments:

1080      EG: PCBF-group, Type(s): Rec

      ele1: Element of EG, Type(s): Tup

      ele2: Element of EG, Type(s): Tup

1085

      Parameters:

      Return Type(s):

      BoolElt

1095      Description:

```

*Returns true if and only if ele1 and ele2 are not equal in EG.*

1100   \*/

**return not** PCBFeq(EG, ele1 , ele2);  
**end function**;

1105   PCBFIsIdentity := **function**(EG, ele)

      /\*

1110    *Arguments:*

*EG: PCBF-group , Type(s): Rec*

*ele: Element of EG, Type(s): Tup*

1115

*Parameters:*

1120    *Return Type(s):*

*BoolElt*

1125    *Description:*

*Returns true if and only if ele is the identity element of EG.*

1130   \*/

**return** PCBFeq(EG, ele , PCBFId(EG));  
**end function**;

1135   PCBFIsId := **function**(EG, ele)

      /\*

1140    *Arguments:*

*EG: PCBF-group , Type(s): Rec*

*ele: Element of EG, Type(s): Tup*

1145

*Parameters:*1150 *Return Type(s):**BoolElt*1155 *Description:**Returns true if and only if ele is the identity element of EG.*1160 *\*/*

```

return PCBFIsIdentity(EG, ele);
end function;

```

1165

PCBFRandom := **function** (EG)*/\**1170 *Arguments:**EG: PCBF-group, Type(s): Rec*1175 *Parameters:**Return Type(s):*1180 *Tup**Description:*1185 *Generates a pseudo-random element of EG.**\*/*

```

g := Random(EG`Q);
1190 n := Random(EG`N);
return PCBFCreateElement(EG, g, n);
end function;

```

```

1195 PCBFRandomElement := function (EG)

    /*

    Arguments :

1200     EG: PCBF-group, Type(s): Rec

    Parameters :

1205

    Return Type(s):

    Tup

1210

    Description :

    Generates a pseudo-random element of EG.

1215    */

    return PCBFRandom(EG);
end function;
1220

PCBFRandomSequence := function (EG, l)

    /*

1225

    Arguments :

    EG: PCBF-group, Type(s): Rec

1230

    Parameters :

    Return Type(s):

1235

    SeqEnum

    Description :

1240

```

*Generates a sequence (of length  $l$ ) of pseudo-random elements of  $EG$ .*

```

*/
1245  randseq := [car<PowerSequence(PowerSequence(IntegerRing())) ,
          EG`N> |];
  for i in [1 .. l] do
    Append(~randseq , PCBFRandom(EG));
  end for;
1250  return randseq;
end function;

```

PCBFRepresentative := **function**(EG)

```

1255  /*
    Arguments:

1260    EG: PCBF-group, Type(s): Rec

    Parameters:

1265    Return Type(s):

    Tup

1270    Description:

    Returns a representative element of EG.

1275  */

  return PCBFId(EG);
end function;

1280 PCBFRep := function(EG)

  /*

1285  Arguments:

    EG: PCBF-group, Type(s): Rec

```

1290 *Parameters:*

*Return Type(s):*

1295 *Tup*

*Description:*

1300 *Returns a representative element of EG.*

*\*/*

**return** PCBFRepresentative(EG);

1305 **end function;**

#### Listing B.6: pcbfarithmetic.m

*/\**

*File:* *pcbfarithmetic.m*

5

*Last modified:* *Fri, 07 Jan 2011 17:21:11 +0000*

*Author(s):* *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

10

*Company:* *University of Warwick <http://www.warwick.ac.uk>*

15 *Description:*

*The methods in this file are used to perform element arithmetic in polycyclic-by-finite groups represented by records of type PCBF.*

20

*Notes:*

*MAGMA V2.16-13.*

25

*Copyright 2006-2010, University of Warwick. All rights reserved.*



```

*/

30  load "pcbfconstruct.m";

forward PCBFConjugateByStrongGen, PCBFConjugateByStrongGenWord;

35  PCBFMult := function(EG, ele1, ele2)

    /*
40    Arguments:

        EG: PCBF-group, Type(s): Rec

45        ele1: Element of EG, Type(s): Tup

        ele2: Element of EG, Type(s): Tup

50    Parameters:

        Return Type(s):

55        Tup

        Description:

60        Computes the product, ele1 * ele2.

    */

    GR := EG`GR;
65    if PCBFrho(EG, ele2) eq Id(GR`G) then
        return <ele1[1], ele1[2] * ele2[2]>; //easy
    end if;
    u := [];
    l := #GR`B;
70    g := Id(GR`G);
    N := EG`N;
    rightword := [];
    for i in [#ele2[1] .. 1 by -1] do
        rightword cat:= ele2[1][i];

```

```

75  end for;
    leftword := [];
    nrightseq := [];
    nrightseq[1] := ele1[2];
    for i in [2 .. 1] do
80      nrightseq[i] := Id(N);
    end for;
    for i in [1 .. 1] do
      g := WordPermutation(GR, ele1[1][i]);
      pt := GR`B[i]^g;
85      while #rightword gt 0 do
        j := Position(GR`bo[i], pt);
        nshift := Id(N);
        if rightword[1] lt 0 then
          pt := GR`B[i]^(g * (GR`S[-rightword[1]])^-1);
          90      k := Position(GR`sgindexlist[i], -rightword[1]);
          leftword cat:= GR`tailinv[i][j][k];
          if GR`utailinv[i][j][k] ne 0 then
            nshift := EG`taileltsinv[GR`utailinv[i][j][k]];
          end if;
95      else
        pt := GR`B[i]^(g * GR`S[rightword[1]]);
        k := Position(GR`sgindexlist[i], rightword[1]);
        leftword cat:= GR`tail[i][j][k];
        if GR`utail[i][j][k] ne 0 then
100      nshift := EG`tailelts[GR`utail[i][j][k]];
        end if;
      end if;
      nrightseq[i] := nshift * PCBFConjugateByStrongGen(EG,
        nrightseq[i], rightword[1]);
      g := SVPermutationInv(GR`G, GR`S, GR`sv[i], GR`bo[i], pt);
105      Remove(~rightword, 1);
    end while;
    u[i] := StrongGenWord(GR, g);
    rightword := leftword;
    leftword := [];
110  end for;
    n := Id(N);
    for i in [1 .. 1 by -1] do
      n := PCBFConjugateByStrongGenWord(EG, n, u[i]) * nrightseq[i];
    end for;
115  return <u, n * ele2[2]>;
end function;

```

PCBFConjugateByStrongGen := **function**(EG, n, i)

```

/*

Arguments:

125   EG: PCBF-group, Type(s): Rec

      n: Element of EG`N, Type(s): GrpPCElt or GrpGPCElt

      i: Integer representing a preimage of a strong generator of
130   the quotient group, EG`Q, Type(s): RngIntElt

Parameters:

135   Return Type(s):

      GrpPCElt or GrpGPCElt

140   Description:

      Returns the conjugate of the element, n, by the element of
      EG represented by i.

145   */

N := EG`N;
if n eq Id(N) then return n; end if;
150 w := Eltseq(n);
ans := Id(N);
for j in [1 .. #w] do
    ans *:= i gt 0 select EG`pcgenconjugates[i][j]^w[j] else EG`
        pcgenconjugatesinv[-i][j]^w[j];
end for;
155 return ans;
end function;

PCBFConjugateByStrongGenWord := function(EG, n, word)

160   /*

Arguments:

165   EG: PCBF-group, Type(s): Rec

      n: Element of EG`N, Type(s): GrpPCElt or GrpGPCElt

```

---

```

170      word: Sequence of integers representing a word in the
      preimages of the strong generators of the quotient group,
      EG`Q, Type(s): SeqEnum

      Parameters:

175

      Return Type(s):

      GrpPCElt or GrpGPCElt

180

      Description:

      Returns the conjugate of the element n by the element of EG
185 represented by word.

      */

      ans := n;
190  for i in [1 .. #word] do
      ans := PCBFConjugateByStrongGen(EG, ans, word[i]);
      end for;
      return ans;
      end function;

195

PCBFMultiply := function(EG, ele1, ele2)

      /*

200  Arguments:

      EG: PCBF-group, Type(s): Rec

      ele1: Element of EG, Type(s): Tup
205 ele2: Element of EG, Type(s): Tup

      Parameters:

210

      Return Type(s):

      Tup
215

```

---

*Description :*

220     *Computes the product  $ele1 * ele2$ .*

*\*/*

**return** PCBFMult(EG, ele1 , ele2);

225 **end function**;

PCBFStar := **function**(EG, ele1 , ele2)

230     */\**

*Arguments :*

*EG: PCBF-group , Type(s): Rec*

235     *ele1: Element of EG, Type(s): Tup*

*ele2: Element of EG, Type(s): Tup*

240     *Parameters :*

*Return Type(s):*

245     *Tup*

*Description :*

250     *Computes the product  $ele1 * ele2$ .*

*\*/*

255     **return** PCBFMult(EG, ele1 , ele2);

**end function**;

PCBFWordMultiply := **function**(EG, posimgs , negimgs , word)

260     */\**

*Arguments :*

```

265      EG: PCBF-group, Type(s): Rec

      posimgs: Sequence of ordered pairs representing elements of
      EG, Type(s): SeqEnum

270      negimgs: Sequence of ordered pairs representing elements of
      EG, Type(s): SeqEnum

      word: Sequence of integers representing a word in the
      elements indexed by posimgs and negimgs, Type(s): SeqEnum
275

      Parameters:

280      Return Type(s):

      Tup

285      Description:

      Computes the product of the sequence of elements of EG
      represented by word.

290      */

      ans := PCBFId(EG);
      for i in [1 .. #word] do
        ans := PCBFMult(EG, ans, word[i] gt 0 select posimgs[word[i]
          ]) else negimgs[-word[i]]);
295      end for;
      return ans;
      end function;

300 PCBFrhoinv := function(EG, g)

      /*

      Arguments:

305      EG: PCBF-group, Type(s): Rec

      g: Element of EG`Q, Type(s): GrpPermElt

310

```

```

Parameters:

Return Type(s):
315   Tup

Description:
320   Returns an inverse of the element, g, of  $EG^*Q$  under the
      natural map from  $EG$  onto  $EG^*Q$ .

  */
325   word := StrongGenWord(EG`GR, g);
      return PCBFWordMultiply(EG, EG`strgenpreimgsnf, EG`
          strgeninvpreimgsnf, word);
end function;

330   PCBFrhoinvSeq := function(EG, elts)

      /*
335   Arguments:

      EG: PCBF-group, Type(s): Rec

      elts: sequence of elements of  $EG^*Q$ , Type(s): SeqEnum
340

Parameters:

345   Return Type(s):

      SeqEnum

350   Description:

      Returns a sequence of preimages in  $EG$  corresponding to the
      elements in elts.

355   */

      return [PCBFrhoinv(EG, elts[i]) : i in [1 .. #elts]];

```

```

end function;

360 PCBFInverse := function(EG, ele)

    /*

365   Arguments:

       EG: PCBF-group, Type(s): Rec

       ele: Element of EG, Type(s): Tup

370   Parameters:

375   Return Type(s):

       Tup

380   Description:

       Computes  $ele^{-1}$ .

       */

385   elt := PCBFrhoInv(EG, PCBFrho(EG, ele)^-1); // ele * elt is an
       element of the normal subgroup used to construct EG
   return PCBFMult(EG, elt, PCBFphiInv(EG, PCBFphi(EG, PCBFMult(
       EG, ele, elt))^-1));
end function;

390 PCBFInv := function(EG, ele)

    /*

395   Arguments:

       EG: PCBF-group, Type(s): Rec

       ele: Element of EG, Type(s): Tup

400   Parameters:
```



```

405  Return Type(s):

      Tup

410  Description:

      Computes  $ele^{-1}$ .

      */
415  return PCBFInverse(EG, ele);
end function;

420 PCBFDivide := function(EG, ele1 , ele2)

      /*

      Arguments:

425      EG: PCBF-group , Type(s): Rec

      ele1: Element of EG, Type(s): Tup

430      ele2: Element of EG, Type(s): Tup

      Parameters:

435      Return Type(s):

      Tup

440      Description:

      Computes  $ele1 * (ele2)^{-1}$ .

445      */

      return PCBFMult(EG, ele1 , PCBFInverse(EG, ele2));
end function;

450 PCBFSlash := function(EG, ele1 , ele2)

```

```

/*
455  Arguments :

      EG: PCBF-group , Type(s): Rec

      ele1: Element of EG, Type(s): Tup
460  ele2: Element of EG, Type(s): Tup

Parameters :
465

Return Type(s):

      Tup
470

Description :

      Computes ele1 * (ele2)^-1.
475
*/

  return PCBFDivide(EG, ele1 , ele2);
end function;
480

PCBFConjugate := function(EG, ele1 , ele2)

/*
485  Arguments :

      EG: PCBF-group , Type(s): Rec

      ele1: Element of EG, Type(s): Tup
490  ele2: Element of EG, Type(s): Tup

Parameters :
495

Return Type(s):

```

500     *Tup*

*Description :*

505     *Computes  $ele1^{ele2} = (ele2)^{-1} * ele1 * ele2$ .*

$*/$

**return** PCBFMult(EG, PCBFMult(EG, PCBFInverse(EG, ele2), ele1),  
                        ele2);

510 **end function**;

PCBFCommutator := **function**(EG, ele1, ele2)

515      $/*$

*Arguments :*

*EG: PCBF-group, Type(s): Rec*

520

*ele1: Element of EG, Type(s): Tup*

*ele2: Element of EG, Type(s): Tup*

525

*Parameters :*

*Return Type(s):*

530

*Tup*

*Description :*

535

*Computes  $(ele1, ele2) = ele1^{-1} * (ele2)^{-1} * ele1 * ele2$ .*

$*/$

540     **return** PCBFMult(EG, PCBFConjugate(EG, PCBFInverse(EG, ele2),  
                        ele1), ele2);

**end function**;

**function** PCBFSeqCommutator(EG, elts)

545

```

/*

Arguments:

550    EG: PCBF-group, Type(s): Rec

        elts: Sequence of elements of EG, Type(s): SeqEnum

555    Parameters:

Return Type(s):

560    Tup

Description:

565    Computes the (left-normed) commutator of the elements in
        elts.

*/

n := #elts;
570  if n eq 2 then
    // base case n = 2
    return PCBFCommutator(EG, elts[1], elts[2]);
  end if;
  // recurse here
575  return PCBFCommutator(EG, PCBFSeqCommutator(EG, elts[1 .. n -
    1]), elts[n]);
end function;

PCBFPower := function(EG, ele, exp)
580
/*

Arguments:

585    EG: PCBF-group, Type(s): Rec

        ele: Element of EG, Type(s): Tup

        exp: Integer, Type(s): RngIntElt

590

```

```

Parameters:

595  Return Type(s):

      Tup

600  Description:

      Computes  $ele^{exp}$ .

      */
605  if exp lt 0 then
      ele := PCBFInverse(EG, ele);
      exp := -exp;
  end if;
610  ans := PCBFIdentity(EG);
  while exp gt 0 do
      if IsOdd(exp) then
          ans := PCBFMult(EG, ans, ele);
      end if;
615  exp := exp div 2;
      ele := PCBFMult(EG, ele, ele);
  end while;
  return ans;
end function;
620

PCBFEleOrder := function(EG, ele)

    /*
625
      Arguments:

      EG: PCBF-group, Type(s): Rec

630  ele: Element of EG, Type(s): Tup

Parameters:

635
      Return Type(s):

      RngIntElt

```

640

*Description :**Computes the order of ele.*

645

\*/

quotord := Order(PCBFrho(EG, ele));

**return** quotord \* Order(PCBFphi(EG, PCBFPower(EG, ele, quotord))**end function;**

## Listing B.7: pcbfcattrans.m

1 /\*

*File :* *pcbfcattrans.m*6 *Last modified:* *Fri, 07 Jan 2011 17:21:11 +0000**Author(s):* *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

11

*Company:* *University of Warwick <http://www.warwick.ac.uk>**Description :*

16

*The methods in this file are used to rewrite polycyclic-by-finite groups represented by records of type PCBF as GrpPC or GrpGPC if such a representation is possible.*

21

*Notes :**MAGMA V2.16-13.*

26

*Copyright 2006-2010, University of Warwick. All rights reserved.*

\*/

31

**load** "pcbfarithmetic.m";

```

PCBFPolycyclicGroup := function(EG)
36
    /*
        Arguments:
41      EG: PCBF-group, Type(s): Rec

        Parameters:

46      Return Type(s):

        GrpPC or GrpGPC, Tup

51      Description:

        Returns a group of type GrpPC or GrpGPC isomorphic to EG.
        Data used to compute the isomorphism and its inverse is
56      returned as a second argument.

        */

    // more rigorous testing to be done!
61    Q, kappa := PCGroup(PCBFQuotGrp(EG)); // kappa: EG`Q --> Q
    N := PCBFNormalSubgrp(EG); // N is of type GrpPC or GrpGPC
    isgrppc := Type(N) eq GrpPC;
    a := PCGenerators(Q);
    b := PCGenerators(N);
66    pcgenpreimgs := [PCBFrhoInv(EG, a[i]@@kappa) : i in [1 .. #a]]
        cat [PCBFphiInv(EG, b[i]) : i in [1 .. #b]];
    pclength := #a + #b;
    F := FreeGroup(pclength);
    c := {@ F.i : i in [1 .. pclength] @} ; // rank is finite so
        this is ok
    powerrelations := []; // power relations for new polycyclic
        presentation
71    conjugaterelations := []; // conjugate relations for new
        polycyclic presentation
    p := [];
    if Order(Q) gt 1 then
        p := PCPrimes(Q);
    end if;
76    m := isgrppc select PCPrimes(N) else PCExponents(N);
    m := p cat m;

```

```

for j in [1 .. #a] do
    // the power and conjugate relations for the pc-generators
    // of the permutation group are computed in this loop
    // isomorphic copies of the polycyclic relations for the
    // normal subgroup are added to R, along with the additional
    // conjugation relations
81 // power relations
    rel := c[j]^m[j] = Id(F);
    rhs := PCBFPower(EG, pcgenpreimgs[j], m[j]);
    for i in [j + 1 .. #a] do
        alpha := Eltseq(kappa(PCBFrho(EG, rhs)))[i];
86 rel := LHS(rel) = RHS(rel) * c[i]^alpha;
        rhs := PCBFMult(EG, PCBFPower(EG, pcgenpreimgs[i], -alpha)
            , rhs);
    end for;
    eleseq := Eltseq(PCBFphi(EG, rhs));
    for i in [Max(j, #a) + 1 .. pclength] do
91 rel := LHS(rel) = RHS(rel) * c[i]^eleseq[i - #a];
    end for;
    Append(~powerrelations, rel);
    // conjugate relations
    for i in [1 .. j - 1] do
96 rel := c[j]^c[i] = Id(F);
        rhs := PCBFConjugate(EG, pcgenpreimgs[j], pcgenpreimgs[i])
            ;
        for k in [i + 1 .. #a] do
            alpha := Eltseq(kappa(PCBFrho(EG, rhs)))[k];
            rel := LHS(rel) = RHS(rel) * c[k]^alpha;
101 rhs := PCBFMult(EG, PCBFPower(EG, pcgenpreimgs[k], -
                alpha), rhs);
        end for;
        eleseq := Eltseq(PCBFphi(EG, rhs));
        for k in [Max(i, #a) + 1 .. pclength] do
            rel := LHS(rel) = RHS(rel) * c[k]^eleseq[k - #a];
106 end for;
        Append(~conjugaterelations, rel);
    end for;
end for;
for j in [#a + 1 .. pclength] do
111 // the power and conjugate relations for the normal subgroup
    // generators are computed in this loop
    // isomorphic copies of the polycyclic relations for the
    // normal subgroup are added to R, along with the additional
    // conjugation relations
    // power relations
    if m[j] ne 0 then
        rel := c[j]^m[j] = Id(F);
116 eleseq := Eltseq(b[j - #a]^m[j]);

```



```

    for i in [j + 1 .. plength] do
        rel := LHS(rel) = RHS(rel) * c[i]^elseq[i - #a];
    end for;
    Append(~powerrelations, rel);
121 end if;
    for i in [1 .. #a] do
        rel := c[j]^c[i] = Id(F);
        rhs := PCBFConjugate(EG, pcgenpreimgs[j], pcgenpreimgs[i])
            ;
        for k in [i + 1 .. #a] do
126         alpha := Eltseq(kappa(PCBFrho(EG, rhs)))[k];
            rel := LHS(rel) = RHS(rel) * c[k]^alpha;
            rhs := PCBFMult(EG, PCBFPower(EG, pcgenpreimgs[k], -
                alpha), rhs);
        end for;
        elseq := Eltseq(PCBFphi(EG, rhs));
131 for k in [#a + 1 .. plength] do
            rel := LHS(rel) = RHS(rel) * c[k]^elseq[k - #a];
        end for;
        Append(~conjugaterelations, rel);
    end for;
136 for i in [#a + 1 .. j - 1] do
        rel := c[j]^c[i] = Id(F);
        elseq := Eltseq(b[j - #a]^b[i - #a]);
        for k in [i + 1 .. plength] do
            rel := LHS(rel) = RHS(rel) * c[k]^elseq[k - #a];
141        end for;
        Append(~conjugaterelations, rel);
        if m[i] eq 0 then
            rel := c[j]^(c[i]^(-1)) = Id(F);
            elseq := Eltseq(b[j - #a]^(b[i - #a]^(-1)));
146        for k in [i + 1 .. plength] do
            rel := LHS(rel) = RHS(rel) * c[k]^elseq[k - #a];
        end for;
        Append(~conjugaterelations, rel);
        end if;
151    end for;
end for;
R := powerrelations cat conjugaterelations;
if isgrppc then
    H, _ := quo<GrpPC : F | R>;
156 else
    H, _ := quo<GrpGPC : F | R >;
end if;
return H, <kappa, pcgenpreimgs>; // the isomorphism kappa is
    needed to compute the isomorphisms between EG and the new
    polycyclic group
end function;

```

161

```
PCBFPolycycliciso := function(EG, P, pcisodata, ele)
```

```
  /*
```

166

```
  Arguments:
```

```
    EG: PCBF-group, Type(s): Rec
```

171

```
    P: Polycyclic group isomorphic to EG, Type(s): GrpPC or GrpGPC
```

```
    pcisodata: Data to compute isomorphism, Type(s): Tup
```

176

```
    ele: Element of EG, Type(s): Tup
```

```
  Parameters:
```

181

```
  Return Type(s):
```

```
    GrpPCElt or GrpGPCElt
```

186

```
  Description:
```

```
    Computes the image of ele (in the polycyclic group isomorphic to EG) as defined by pcisodata.
```

191

```
  */
```

```
  kappa := pcisodata[1];
```

```
  Q := Codomain(kappa);
```

196

```
  pcgenpreimgs := pcisodata[2];
```

```
  alpha := [];
```

```
  t := 0;
```

```
  if Order(Q) gt 1 then
```

```
    t := #Eltseq(Id(Q));
```

201

```
  end if;
```

```
  for i in [1 .. t] do
```

```
    alpha[i] := Eltseq(kappa(PCBFRho(EG, ele)))[i];
```

```
    ele := PCBFMult(EG, PCBFPower(EG, pcgenpreimgs[i], -alpha[i]), ele);
```

```
  end for;
```

206

```
  alpha cat:= Eltseq(PCBFphi(EG, ele));
```

```
  return P ! alpha;
```

```

end function;

211 PCBFPolycyclicisoSeq := function(EG, P, pcisodata, elts)

    /*

    Arguments:

216     EG: PCBF-group, Type(s): Rec

        P: Polycyclic group isomorphic to EG, Type(s): GrpPC or
        GrpGPC

221     pcisodata: Data to compute isomorphism, Type(s): Tup

        elts: Sequence of elements of EG, Type(s): SeqEnum

226     Parameters:

        Return Type(s):

231     SeqEnum

        Description:

236     Returns a sequence of the respective images (in P) of each
        element of elts.

    */

241 return [PCBFPolycycliciso(EG, P, pcisodata, elts[i]) : i in [1
    .. #elts]];
end function;

246 PCBFPolycyclicisoinv := function(EG, pcisodata, y)

    /*

    Arguments:

251     EG: PCBF-group, Type(s): Rec

        pcisodata: Data to compute isomorphism, Type(s): Tup

```

256     *y: Element of identified polycyclic group isomorphic to EG,*  
        *Type(s): GrpPCElt or GrpGPCElt*

*Parameters:*

261

*Return Type(s):*

*Tup*

266

*Description:*

*Computes the image of y in EG as defined by pcisodata.*

271

*\*/*

ans := PCBFId(EG);

alpha := Eltseq(y);

276

**for** i **in** [1 .. #alpha] **do**

    ans := PCBFMult(EG, ans, PCBFPower(EG, pcisodata[2][i],  
                     alpha[i]));

**end for**;

**return** ans;

**end function**;

281

PCBFPolycyclicisoinvSeq := **function**(EG, pcisodata, pcelts)

*/\**

286

*Arguments:*

*EG: PCBF-group, Type(s): Rec*

291

*pcisodata: Data to compute isomorphism, Type(s): Tup*

*y: Sequence of elements of the polycyclic group isomorphic  
    to EG, Type(s): SeqEnum*

296

*Parameters:*

*Return Type(s):*

301

*SeqEnum**Description:*

306

*Returns a sequence of the respective images (in EG) of each element of y.*

\*/

311

```
return [PCBFPolycyclicisoinv(EG, pcisodata, pcelts[i]) : i in
        [1 .. #pcelts]];
end function;
```

## Listing B.8: pcbfschreiersims.m

/\*

2

*File:* *pcbfschreiersims.m*

*Last modified:* *Sun, 06 Feb 2011 22:14:13 +0000*

7

*Author(s):* *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

12 *Company:* *University of Warwick <<http://www.warwick.ac.uk>>*

*Description:*

17 *The methods in this file allow one to work with the permutation action (induced by the quotient group) of a polycyclic-by-finite group represented by a record of type PCBF.*

22 *Notes:*

*MAGMA V2.16–13.*

27 *Copyright 2006–2010, University of Warwick. All rights reserved.*

\*/

```

32 load "pcbfcattrans.m";

PCBFQuotAction := function(EG, alpha, ele)

37  /*

    Arguments:

        EG: PCBF-group, Type(s): Rec
42
        alpha: Element of the set upon which EG`Q acts, Type(s):
            Elt

        ele: Element of EG, Type(s): Tup
47

    Parameters:

52  Return Type(s):

        Elt

57  Description:

        Computes the action of the first component of ele on the
        point alpha.

62  */

word := [];
S := EG`GR`S;
for i in [#ele[1] .. 1 by -1] do
67   word cat:= ele[1][i];
end for;
gamma := alpha;
for i in [1 .. #word] do
   gamma := word[i] gt 0 select gamma^(S[word[i]]) else gamma
   ^((S[-word[i]])^-1);
72 end for;
return gamma;
end function;

77 PCBFQuotOrbit := function(EG, alpha, X)

```

```

/*

Arguments:

82   EG: PCBF-group, Type(s): Rec

      alpha: Element of the set upon which EG`Q acts, Type(s):
      Elt

87   X: Sequence of ordered pairs representing elements of EG,
      Type(s): SeqEnum

92   Parameters:

      Return Type(s):

97   SeqEnum, SeqEnum

Description:

102   Computes the orbit of the point alpha under the action of
      X . The function returns the orbit and a Schreier
      vector
      corresponding to the orbit with indexing relative to the
      set X.

107  */

r := #X;
      v := [0];
orb := [alpha];
112 orbsize := 1;
c := 1;
while c le orbsize do
  pt := orb[c];
  for i in [1 .. r] do
117   ppt := PCBFQuotAction(EG, pt, X[i]);
      if ppt notin orb then
        Append(~orb, ppt);
        orbsize += 1;
        v[orbsize] := i;
122   end if;
  end for;
  c += 1;

```

```

    end while;
    return v, orb;
127 end function;

PCBFQuotSVWordInv := function(EG, Ypos, Yneg, v, orb, pt)

132 /*

    Arguments:

        EG: PCBF-group, Type(s): Rec

137        Ypos: Sequence of ordered pairs representing elements of
            EG, Type(s): SeqEnum

        Yneg: Sequence of ordered pairs representing elements of
142 EG, Type(s): SeqEnum

        v: Sequence of integers representing a Schreier vector
            (with indexing relative to Ypos and Yneg), Type(s): SeqEnum

147        orb: Sequence containing exactly the orbit of orb[1] under
            Y, Type(s): SeqEnum

        pt: Element of the set upon which EG`Q acts, Type(s): Elt

152

    Parameters:

        Return Type(s):

157        SeqEnum

    Description:

162        Returns inverse of what PCBFQuotSVWord should return, as an
            integer sequence i.e. a word in strong generators taking
            base point to pt as defined by the Schreier vector v.

167 */

    pos := Position(orb, pt);
    r := v[pos];
    w := [];
172 while r ne 0 do

```



```

Append(~w, -r);
y := r gt 0 select PCBFInverse(EG, Ypos[r]) else Yneg[-r];
pt := PCBFQuotAction(EG, pt, y);
pos := Position(orb, pt);
177   r := v[pos];
end while;
return WordInverse(w);
end function;

182
PCBFQuotSVElement := function(EG, Ypos, Yneg, v, orb, pt)

    /*

187   Arguments:

        EG: PCBF-group, Type(s): Rec

        Ypos: Sequence of ordered pairs representing elements of
192   EG, Type(s): SeqEnum

        Yneg: Sequence of ordered pairs representing elements of
        EG, Type(s): SeqEnum

197   v: Sequence of integers representing a Schreier vector
        (with indexing relative to Ypos and Yneg), Type(s): SeqEnum

        orb: Sequence containing exactly the orbit of orb[1] under
        Y, Type(s): SeqEnum

202   pt: Element of the set upon which EG`Q acts, Type(s): Elt


        Parameters:

207

        Return Type(s):

        Tup

212

        Description:

        Returns the element of EG in normal form taking pt to base
217   point as defined by the Schreier vector v.

        */

```

```

pos := Position(orb, pt);
222 r := v[pos];
    u := PCBFId(EG);
    y := PCBFId(EG);
    while r ne 0 do
        y := r gt 0 select PCBFInverse(EG, Ypos[r]) else Yneg[-r];
227 u := PCBFMult(EG, u, y);
        pt := PCBFQuotAction(EG, pt, y);
        pos := Position(orb, pt);
        r := v[pos];
    end while;
232 return u;
end function;

```

```

PCBFQuotSVElementInv := function(EG, Ypos, Yneg, v, orb, pt)
237
    /*
        Arguments:

242    EG: PCBF-group, Type(s): Rec

        Ypos: Sequence of ordered pairs representing elements of
        EG, Type(s): SeqEnum

247    Yneg: Sequence of ordered pairs representing elements of
        EG, Type(s): SeqEnum

        v: Sequence of integers representing a Schreier vector
        (with indexing relative to Ypos and Yneg), Type(s): SeqEnum
252

        orb: Sequence containing exactly the orbit of orb[1] under
        Y, Type(s): SeqEnum

        pt: Element of the set upon which EG`Q acts, Type(s): Elt
257

        Parameters:

262    Return Type(s):

        Tup

267    Description:

```

*Returns the element of EG in normal form taking base point to pt as defined by the Schreier vector v.*

272     \*/

```

return PCBFInverse(EG, PCBFQuotSVElement(EG, Ypos, Yneg, v,
      orb, pt));
end function;

```

277

```
PCBFQuotStrip := function(EG, ele, B, S, Sinv, sv, bo)
```

/\*

282     *Arguments:*

*EG: PCBF-group, Type(s): Rec*

*ele: Element of EG, Type(s): Tup*

287

*B: Base of EG`Q, Type(s): SeqEnum*

*S: 2-D sequence of ordered pairs representing elements of EG, Type(s): SeqEnum*

292

*Sinv: 2-D sequence of ordered pairs representing elements of EG, Type(s): SeqEnum*

297     *sv: Sequence of integer sequences each representing a Schreier vector (with indexing relative to the sequences of S and Sinv), Type(s): SeqEnum*

*bo: Sequence containing exactly the orbits of each base point under S, Type(s): SeqEnum*

302

*Parameters:*

307     *Return Type(s):*

*Tup, RngIntElt*

312     *Description:*

*Strips the element ele as if it were an element of the quotient permutation group using the "base" B and "strong*

```

    generating set" represented by the indexed set S.
317  */

    // bo[i] is the i-th basic orbit, and sv[i] is the Schreier
    vector corresponding to this orbit and the generating set
    specified by S[i]
    h := ele;
322  l := #B;
    for i in [1 .. l] do
        // h fixes base points B[1], B[2], ..., B[i-1]
        pt := PCBFQuotAction(EG, B[i], h);
        if pt notin bo[i] then
327         return h, i;
        end if;
        u := PCBFQuotSVElement(EG, S[i], Sinv[i], sv[i], bo[i], pt);
        h := PCBFMult(EG, h, u);
    end for;
332  return h, l + 1;
end function;

PCBFDepth := function(EG, B, ele)
337
    /*

    Arguments:

342    EG: PCBF-group, Type(s): Rec

        B: Sequence of elements of EG`GR`natset, Type(s): SeqEnum

        ele: Element of EG, Type(s): Tup
347

    Parameters:

352    Return Type(s):

        RngIntElt

357    Description:

        Returns the integer k such that ele fixes B[1], B[2], ...,
        B[k-1] and moves B[k].

```

```

362    */

    for k in [1 .. #B] do
        if PCBFQuotAction(EG, B[k], ele) ne B[k] then return k; end
        if;
    end for;
367    return #B + 1;
end function;

PCBFFindNewBasePoint := function(EG, Omega, ele)
372
    /*

    Arguments:

377    EG: PCBF-group, Type(s): Rec

        Omega: Non-empty sequence of elements of EG`GR`natset
        containing a base for EG`GR`G, Type(s): SeqEnum

382    ele: Element of EG with non-trivial first component,
        Type(s): Tup

    Parameters:

387

    Return Type(s):

        Elt

392

    Description:

        Returns a point in Omega that is moved by ele.

397
    */

    i := 1;
    while PCBFQuotAction(EG, Omega[i], ele) eq Omega[i] do
402        i += 1;
    end while;
    return Omega[i];
end function;

407
PCBFStandardSchreierSims := function(EG, gens)

```

```

/*
412  Arguments :

      EG: PCBF-group, Type(s): Rec

      gens: Sequence of ordered pairs representing elements of
417  EG, Type(s): SeqEnum

Parameters :

422  Return Type(s):

      SeqEnum, SetIndx, SeqEnum

427  Description :

      Returns a "base" and "strong generating set" for the
      subgroup of EG generated by gens, together with a
432  generating sequence for the intersection of this subgroup
      with the image of EG`N in EG.

*/

437  B := []; // new base
      Omega := EG`GR`B;
      G := EG`GR`G;
      k := 0;
      S := [];
442  Sinv := [];
      bo := [];
      sv := [];
      strippedgens := [];
      mixedgens := [];
447  for i in [1 .. #gens] do
      if PCBFrho(EG, gens[i]) ne Id(G) then
        Append(~mixedgens, gens[i]);
      elif not PCBFIsId(EG, gens[i]) then
        Append(~strippedgens, gens[i]);
452  end if;
      end for;
      strippedgensinv := [PCBFInv(EG, x) : x in strippedgens];
      depth := [];
      for i in [1 .. #mixedgens] do

```

```

457   depth[i] := PCBFDepth(EG, B, mixedgens[i]);
      if depth[i] eq k + 1 then
        Append(~B, PCBFFindNewBasePoint(EG, Omega, mixedgens[i]));
        k += 1;
      end if;
462   end for;
      for i in [1 .. k] do
        S[i] := [mixedgens[j] : j in [1 .. #mixedgens] | depth[j] ge
          i];
        Sinv[i] := [PCBFInv(EG, x) : x in S[i]];
        S[i] cat:= strippedgens;
467   Sinv[i] cat:= strippedgensinv;
        sv[i], bo[i] := PCBFQuotOrbit(EG, B[i], S[i]);
      end for;
      NH := sub<EG`N | [PCBFphi(EG, x) : x in strippedgens]>;
      i := k;
472   uptodate := true;
      while i ge 1 do
        for pt in bo[i] do
          u1 := PCBFQuotSVElementInv(EG, S[i], Sinv[i], sv[i], bo[i]
            ], pt);
          for x in S[i] do
477   pt2 := PCBFQuotAction(EG, pt, x);
          u2 := PCBFQuotSVElement(EG, S[i], Sinv[i], sv[i], bo[i],
            pt2);
          ele := PCBFMult(EG, PCBFMult(EG, u1, x), u2);
          if PCBFIsId(EG, ele) then continue x; end if;
          uptodate := true;
482   ele, j := PCBFQuotStrip(EG, ele, B, S, Sinv, sv, bo);
          if PCBFrho(EG, ele) ne Id(G) then
            // new strong generator at level j
            uptodate := false;
            if j gt k then
487   // extend base
              Append(~B, PCBFFindNewBasePoint(EG, Omega, ele));
              k += 1;
              S[k] := strippedgens; // maintain inclusion
              Sinv[k] := strippedgensinv; // maintain inclusion
            end if;
492   elif PCBFphi(EG, ele) notin NH then
            uptodate := false;
            Append(~strippedgens, ele);
            Append(~strippedgensinv, PCBFInv(EG, ele));
497   NH := sub<EG`N | NH, PCBFphi(EG, ele)>;
            j := k;
          end if;
          if not uptodate then
            for t in [i + 1 .. j] do

```

```

502         Append(~S[t], ele);
           Append(~Sinv[t], PCBFInv(EG, ele));
           sv[t], bo[t] := PCBFQuotOrbit(EG, B[t], S[t]);
           end for;
           i := j + 1;
507         break pt;
           end if;
           end for;
           end for;
           i -= 1;
512     end while;
     SGS := {@@};
     for i in [1 .. #S] do
       for j in [1 .. #S[i]] do
         if PCBFrho(EG, S[i][j]) ne Id(G) then
517           Include(~SGS, S[i][j]);
         end if;
       end for;
     end for;
     return B, SGS, Generators(NH);
522 end function;

PCBFNCLSchreierSims := function(EG, gens)

527  /*

    Arguments:

    EG: PCBF-group, Type(s): Rec

532    gens: Sequence of ordered pairs representing elements of
          EG, Type(s): SeqEnum

537  Parameters:

    Return Type(s):

542    SetIndx, SeqEnum

    Description:

547    Returns a "base" and "strong generating set" for the
          subgroup of EG generated by gens, together with a
          generating sequence for the intersection of this subgroup

```



*with the image of  $EG \backslash N$  in  $EG$ .*

```

552  */

      B := []; // new base
      Omega := EG`GR`B;
      G := EG`GR`G;
557  k := 0;
      S := [];
      Sinv := [];
      bo := [];
      sv := [];
562  depth := [];
      strippedgens := [];
      mixedgens := [];
      for i in [1 .. #gens] do
        if PCBFrho(EG, gens[i]) ne Id(G) then
567          Append(~mixedgens, gens[i]);
        elif not PCBFIsId(EG, gens[i]) then
          Append(~strippedgens, gens[i]);
        end if;
      end for;
572  depth := [];
      for i in [1 .. #mixedgens] do
        depth[i] := PCBFDepth(EG, B, mixedgens[i]);
        if depth[i] eq k + 1 then
          Append(~B, PCBFFindNewBasePoint(EG, Omega, mixedgens[i]));
577          k += 1;
        end if;
      end for;
      for i in [1 .. k] do
        S[i] := [mixedgens[j] : j in [1 .. #mixedgens] | depth[j] ge
          i];
582      Sinv[i] := [PCBFInv(EG, x) : x in S[i]];
        sv[i], bo[i] := PCBFQuotOrbit(EG, B[i], S[i]);
      end for;
      i := k;
      while i ge 1 do
587      for pt in bo[i] do
        u1 := PCBFQuotSVElementInv(EG, S[i], Sinv[i], sv[i], bo[i]
          ], pt);
        for x in S[i] do
          pt2 := PCBFQuotAction(EG, pt, x);
          u2 := PCBFQuotSVElement(EG, S[i], Sinv[i], sv[i], bo[i],
            pt2);
592          ele := PCBFMult(EG, PCBFMult(EG, u1, x), u2);
          if PCBFIsId(EG, ele) then continue x; end if;
          ele, j := PCBFQuotStrip(EG, ele, B, S, Sinv, sv, bo);

```

```

        if PCBFrho(EG, ele) ne Id(G) then
            // new strong generator at level j
597         if j gt k then
            // extend base
            Append(~B, PCBFFindNewBasePoint(EG, Omega, ele));
            k += 1;
            S[k] := [];
602         Sinv[k] := [];
        end if;
        for t in [i + 1 .. j] do
            Append(~S[t], ele);
            Append(~Sinv[t], PCBFInv(EG, ele));
607         sv[t], bo[t] := PCBFQuotOrbit(EG, B[t], S[t]);
        end for;
        i := j + 1;
        break pt;
        elif not PCBFIsId(EG, ele) then
612         Include(~strippedgens, ele);
        end if;
    end for;
end for;
    i -= 1;
617 end while;
SGS := {@@};
for i in [1 .. #S] do
    for j in [1 .. #S[i]] do
        Include(~SGS, S[i][j]);
622    end for;
end for;
// find normal closure of strippedgens in the subgroup
// generated by gens
NHgens := [PCBFphi(EG, x) : x in strippedgens];
NH := sub<EG`N | NHgens>;
627 closurefound := false;
while not closurefound do
    for a in NHgens do
        for ele in mixedgens do
            apreimg := PCBFphiinv(EG, a);
632            newconj := PCBFphi(EG, PCBFConjugate(EG, apreimg, ele));
            if newconj notin NH then
                NH := sub<EG`N | NH, newconj>;
            end if;
        end for;
    end for;
637 end for;
closurefound := true;
NHgens := Generators(NH);
// test to see if normal closure has been found
for a in NHgens do

```

```

642     for ele in mixedgens do
        apreimg := PCBFphiinv(EG, a);
        newconj := PCBFphi(EG, PCBFConjugate(EG, apreimg, ele));
        if newconj notin NH then
            // conjugates still missing
647         closurefound := false;
            break a;
        end if;
    end for;
    end for;
652 end while;
    return B, SGS, NHgens;
end function;

657 PCBFNCLKeepBaseSchreierSims := function(EG, gens)

    /*

    Arguments:

662     EG: PCBF-group, Type(s): Rec

        gens: Sequence of ordered pairs representing elements of
        EG, Type(s): SeqEnum

667

    Parameters:

672     Return Type(s):

        SetIndx, SeqEnum

677     Description:

        Returns a "base" and "strong generating set" for the
        subgroup of EG generated by gens, together with a
        generating sequence for the intersection of this subgroup
682     with the image of EG`N in EG. The base returned is the same
        as that for EG`GR`G.

    */

687 B := EG`GR`B; // base to be used in Schreier—Sims procedure
    G := EG`GR`G;

```

```

k := #B; // this does not change as it is known that no
        element of EG (with nontrivial first entry) can fix all of
        the elements of B
S := [];
Sinv := [];
692 bo := [];
sv := [];
strippedgens := [];
mixedgens := [];
depth := [];
697 for i in [1 .. #gens] do
    if PCBFrho(EG, gens[i]) ne Id(G) then
        Append(~mixedgens, gens[i]);
        depth[#mixedgens] := PCBFDepth(EG, B, gens[i]);
    elif not PCBFIsId(EG, gens[i]) then
702     Append(~strippedgens, gens[i]);
    end if;
end for;
for i in [1 .. k] do
    S[i] := [mixedgens[j] : j in [1 .. #mixedgens] | depth[j] ge
              i];
707     Sinv[i] := [PCBFInv(EG, x) : x in S[i]];
    sv[i], bo[i] := PCBFQuotOrbit(EG, B[i], S[i]);
end for;
i := k;
while i ge 1 do
712     for pt in bo[i] do
        u1 := PCBFInverse(EG, PCBFQuotSVElement(EG, S[i], Sinv[i],
            sv[i], bo[i], pt));
        for x in S[i] do
            pt2 := PCBFQuotAction(EG, pt, x);
            u2 := PCBFQuotSVElement(EG, S[i], Sinv[i], sv[i], bo[i],
                pt2);
717         ele := PCBFMult(EG, PCBFMult(EG, u1, x), u2);
            if PCBFIsId(EG, ele) then continue x; end if;
            ele, j := PCBFQuotStrip(EG, ele, B, S, Sinv, sv, bo);
            if j le k then
                // new strong generator ele at level j
722             for t in [i + 1 .. j] do
                Append(~S[t], ele);
                Append(~Sinv[t], PCBFInv(EG, ele));
                sv[t], bo[t] := PCBFQuotOrbit(EG, B[t], S[t]);
            end for;
727             i := j + 1;
            break pt;
            elif not PCBFIsId(EG, ele) then
                Include(~strippedgens, ele);
            end if;

```

```

732     end for;
    end for;
    i -= 1;
  end while;
  SGS := {@@};
737  for i in [1 .. #S] do
    for j in [1 .. #S[i]] do
      Include(~SGS, S[i][j]);
    end for;
  end for;
  NHgens := [PCBFphi(EG, x) : x in strippedgens];
  NH := sub<EG`N | NHgens>;
  // find normal closure of NH in subgroup generated by gens
  closurefound := false;
  while not closurefound do
747    for a in NHgens do
      for ele in mixedgens do
        apreimg := PCBFphiinv(EG, a);
        newconj := PCBFphi(EG, PCBFConjugate(EG, apreimg, ele));
        if newconj notin NH then
752          NH := sub<EG`N | NH, newconj>;
        end if;
      end for;
    end for;
    closurefound := true;
757    NHgens := Generators(NH);
    // test to see if normal closure has been found
    for a in NHgens do
      for ele in mixedgens do
        apreimg := PCBFphiinv(EG, a);
762        newconj := PCBFphi(EG, PCBFConjugate(EG, apreimg, ele));
        if newconj notin NH then
          // conjugates still missing
          closurefound := false;
          break a;
767        end if;
      end for;
    end for;
  end while;
  return B, SGS, NHgens;
772 end function;

```

PCBFsubBasicOrbits := function(EG, B, S)

777 /\*

Arguments:

```

782      EG: PCBF-group, Type(s): Rec

      B: Base of the corresponding quotient group of the subgroup
        of EG generated by S, Type(s): SeqEnum

      S: Sequence of ordered pairs representing elements of EG,
787      Type(s): SetIndx

Parameters:

792      Return Type(s):

      SeqEnum, SeqEnum

797      Description:

      Computes the basic orbits and Schreier vectors
      (relative to B and S) of the corresponding quotient group
802      of the subgroup of EG generated by S. This function is no
        longer used.

      */

807      l := #B;
      if l eq 0 then return [], []; end if;
      m := #S;
      X := Isetseq(S);
      rhoX := [PCBFrho(EG, X[i]) : i in [1 .. m]];
812      Gsub := sub<EG`GR`G | rhoX>;
      sv := [];
      bo := [];
      bs := [Stabiliser(Gsub, B[1 .. i - 1]) : i in [1 .. l]];
      sv[1], bo[1] := PCBFQuotOrbit(EG, B[1], X);
817      for i in [2 .. l] do
        indexmap := [k : k in [1 .. m] | rhoX[k] in bs[i]];
        sv[i], bo[i] := PCBFQuotOrbit(EG, B[i], X[indexmap]);
        for j in [2 .. #sv[i]] do
          sv[i][j] := indexmap[sv[i][j]];
822      end for;
      end for;
      return sv, bo;
end function;

```

```

/*

File:                                pcbfsubgroup.m

5 Last modified:                      Sun, 06 Feb 2011 22:14:13 +0000

Author(s):                           Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>
10

Company:                             University of Warwick <http://www.warwick.ac.
uk>

15 Description:

The methods in this file are used to construct subgroups of
polycyclic-by-finite groups represented by records of type PCBF.

20

Notes:

MAGMA V2.16-13.

25

Copyright 2006-2010, University of Warwick. All rights reserved.

*/

30 BATCHSIZE := 10;

load "pcbfschreiersims.m";

35

forward PCBFiotaSeq, PCBFiota, PCBFiotainv,
PCBFCleanGeneratingSet, PCBFsubConstruct;

40 function PCBFsub(EG, gens)

/*

Arguments:

45 EG: PCBF-group, Type(s): Rec

```

```

    gens: Sequence of ordered pairs representing elements of
    EG, Type(s): SeqEnum or SetIndx
50

    Parameters:

55    Return Type(s):

        Rec

60    Description:

        Returns a PCBF record representing the subgroup of EG
        generated by the elements in gens.

65    */

    if not EG`ismaster then
        // subgroups are only constructed as subgroups of a master
        group
        gens := PCBFiotaSeq(EG, gens);
70    EG := EG`supergrp; // master group
        return PCBFsub(EG, gens);
    end if;
    gens := PCBFcleanGeneratingSet(EG, gens);
    B, SGS, LHgens := PCBFNCLSchreierSims(EG, gens);
75    sgseq := Isetseq(SGS);
    sgseq, pointer := PCBFcleanGeneratingSet(EG, sgseq);
    if pointer le #sgseq then
        LHgens cat:= sgseq[pointer .. #sgseq];
    end if;
80    strgenpreimgs := {@@}; // indexed set to correspond with
        definiton
    S := {@@}; // indexed set to correspond with record definition
    for i in [1 .. pointer - 1] do
        Include(~strgenpreimgs, sgseq[i]); // preimages for strong
        generators in transversal
        Include(~S, PCBFrho(EG, sgseq[i])); // strong generators for
        permutation bit of subgroup
85    end for;
    QH := sub<EG`Q | S>;
    sv, bo := BasicOrbitsSV(QH, B, S);
    GR := GrpPermData(QH, B, S, sv, bo);
    LH := sub<EG`N | LHgens>;
90    return PCBFsubConstruct(EG, LH, GR, strgenpreimgs);

```



**end function ;**

PCBFCleanGeneratingSet := **function**(EG, gens)

95

/\*

*Arguments :*

100

*EG: PCBF-group, Type(s): Rec*

*gens: Sequence of ordered pairs representing elements of  
EG, Type(s): SeqEnum*

105

*Parameters :*

*Return Type(s):*

110

*SeqEnum, RngIntElt*

*Description :*

115

*Returns a sequence which contains no redundant generators  
and generates the same subgroup in EG as gens.*

*\*/*

gensl := [car<PowerSequence(PowerSequence(IntegerRing()))], EG`  
N> | ]; // the mixed generators

120

gensr := [car<PowerSequence(PowerSequence(IntegerRing()))], EG`  
N> | ]; // these elements map to the trivial element in the  
quotient group

**for** i **in** [1 .. #gens] **do**

*// nonredundant generating set created from gens in one pass*

**if** PCBFRho(EG, gens[i]) **eq** Id(EG`GR`G) **then**

*// gens[i] maps to the identity in the quotient group*

125

**if** gens[i][2] **ne** Id(EG`N) **then**

*// add nontrivial generator to gensr*

Include(~gensr, gens[i]);

**end if;**

**else**

130

*// check to see whether gens[i] has the same image in the  
quotient group as any previously added generator*

j := 1;

**while** j **le** #gensl **do**

**if** gens[i][1] **eq** gensl[j][1] **then**

```

    ele := PCBFMult(EG, gens[i], PCBFInverse(EG, gensl[j]))
    );
135    // ele maps to the identity in the quotient group
    // hence it can be added to gensr
    if ele[2] ne Id(EG`N) then
    // add nontrivial generator to gensr
    Include(~gensr, ele);
    end if;
140    break; // throw gens[i] away
    else
    // continue checking
    j += 1;
    end if;
145    end while;
    if j gt #gensl then Append(~gensl, gens[i]); end if; //
    genuine new generator
    end if;
    end for;
    return gensl cat gensr, #gensl + 1;
150 end function;

```

PCBFsubConstruct := **function**(EG, LH, GR, strgenpreimgs)

155 /\*

*Arguments:*

*EG: PCBF-group, Type(s): Rec*

160

*LH: Intersection of the subgroup of EG generated by S and  
the image of EG`N in EG, Type(s): GrpPC or GrpGPC*

165

*GR: Record holding data for corresponding quotient group of  
the subgroup of EG to be constructed as a permutation  
group, Type(s): Rec*

170 *strgenpreimgs: Indexed set containing preimages (in EG) of  
the strong generators for the corresponding quotient group  
of the subgroup of EG to be constructed, Type(s): SetIndx*

*Parameters:*

175 *Return Type(s):*

*Rec*

180 *Description:*

*Mimics the construction in PCBFMasterConstruct to construct a PCBF record representing the required subgroup of EG.*

185 *\*/*

```

EGsub := rec<PCBF | >;
EGsub`E := EG`E;
EGsub`GR := GR;
190 EGsub`Q := GR`G;
EGsub`N := LH;
EGsub`supergrp := EG;
EGsub`strgenpreimgs := strgenpreimgs;
m := #strgenpreimgs;
195 strgeninvpreimgs := {@@};
for i in [1 .. m] do
    Include(~strgeninvpreimgs, (PCBFElOrder(EG, strgenpreimgs[i]
        ]) eq 2) select strgenpreimgs[i] else PCBFInv(EG,
        strgenpreimgs[i]));
end for;
EGsub`strgeninvpreimgs := strgeninvpreimgs;
200 e := <StrongGenNormalForm(GR, Id(GR`G)), Id(LH)>;
EGsub`e := e;
pcgens := PCGenerators(LH);
r := #pcgens;
pcgenconjugates := [];
205 pcgenconjugatesinv := [];
for i in [1 .. m] do
    pcgenconjugates[i] := [LH | ];
    pcgenconjugatesinv[i] := [LH | ];
    xim := strgenpreimgs[i];
210 for j in [1 .. r] do
        // computing conjugates of the jth polycyclic generator
        with the ith preimage of the strong generating set of Q
        and its inverse
        yim := PCBFphiinv(EG, EG`N ! pcgens[j]);
        pcgenconjugates[i][j] := LH ! PCBFConjugate(EG, yim,
            strgenpreimgs[i])[2]; // stored as an element of LH
        pcgenconjugatesinv[i][j] := LH ! PCBFConjugate(EG, yim,
            strgeninvpreimgs[i])[2]; // stored as an element of LH
215 end for;
end for;
l := #GR`B; // number of elements in the stored base of the
        quotient group
S := GR`S;
tail := GR`tail;

```

```

220  tailinv := GR`tailinv;
    utail := GR`utail;
    utailinv := GR`utailinv;
    h := [];
    g := [];
225  w := [];
    tailelts := [LH | ];
    taileltsinv := [LH | ];
    for i in [1 .. l] do
        orb := GR`bo[i];
230  v := GR`sv[i];
        sg := GR`sgindexlist[i];
        for j in [1 .. #orb] do
            pt := orb[j];
            for k in [1 .. #sg] do
235  g := SVWordInv(GR`G, S, v, orb, pt);
                if utail[i][j][k] ne 0 then
                    ipt := pt^S[sg[k]];
                    h := SVWordInv(GR`G, S, v, orb, ipt);
                    ux := PCBFMult(EG, PCBFWordMultiply(EG, strgenpreimgs,
                        strgeninvpreimgs, g), strgenpreimgs[sg[k]]);
240  th := PCBFMult(EG, PCBFWordMultiply(EG, strgenpreimgs,
                        strgeninvpreimgs, tail[i][j][k]), PCBFWordMultiply
                        (EG, strgenpreimgs, strgeninvpreimgs, h));
                    tailelts[utail[i][j][k]] := LH ! PCBFphi(EG, PCBFMult(
                        EG, PCBFInv(EG, th), ux));
                end if;
                if utailinv[i][j][k] ne 0 then
                    ipt := pt^(S[sg[k]]^-1);
245  h := SVWordInv(GR`G, S, v, orb, ipt);
                    uxinv := PCBFMult(EG, PCBFWordMultiply(EG,
                        strgenpreimgs, strgeninvpreimgs, g),
                        strgeninvpreimgs[sg[k]]);
                    th := PCBFMult(EG, PCBFWordMultiply(EG, strgenpreimgs,
                        strgeninvpreimgs, tailinv[i][j][k]),
                        PCBFWordMultiply(EG, strgenpreimgs,
                        strgeninvpreimgs, h));
                    taileltsinv[utailinv[i][j][k]] := LH ! PCBFphi(EG,
                        PCBFMult(EG, PCBFInv(EG, th), uxinv));
                end if;
250  end for;
            end for;
        end for;
    end for;
    EGsub`pcgens := pcgens;
    EGsub`pcgenconjugates := pcgenconjugates;
255  EGsub`pcgenconjugatesinv := pcgenconjugatesinv;
    EGsub`tailelts := tailelts;
    EGsub`taileltsinv := taileltsinv;

```

```

EGsub`ismaster := false;
EGsub`pcgensnf := [];
260 for i in [1 .. #pcgens] do
    EGsub`pcgensnf[i] := <e[1], pcgens[i]>;
end for;
EGsub`strgenpreimsgsnf := [];
EGsub`strgeninvpreimsgsnf := [];
265 for i in [1 .. #strgenpreimsgs] do
    -, EGsub`strgenpreimsgsnf[i] := PCBFiotainv(EGsub,
        strgenpreimsgs[i]);
    -, EGsub`strgeninvpreimsgsnf[i] := PCBFiotainv(EGsub,
        strgeninvpreimsgs[i]);
end for;
EGsub`grpgens := EGsub`strgenpreimsgsnf cat EGsub`pcgensnf;
270 return EGsub;
end function;

PCBFiotainv := function(EGsub, ele)
275
    /*

    Arguments:

280    EGsub: PCBF-group, Type(s): Rec

    ele: Ordered pair representing an element of
    EGsub`supergrp, Type(s): Tup

285

    Parameters:

    Return Type(s):

290    BoolElt, Tup

    Description:

295    Tests to see if ele is an element of EGsub and if it is,
    returns true and the normal form of ele in the subgroup.
    ele is assumed to be a member of the super group of EGsub.
    Some of this code overlaps with the PCBFQuotStrip function.
300    Do NOT use this method directly. Use PCBFCoerce instead.

    */

```

```

    if EGsub`ismaster then return true, ele; end if;
305 EG := PCBFSSuperGrp(EGsub);
    h := ele; // element of EG
    B := EGsub`GR`B;
    strgenpreimgs := EGsub`strgenpreimgs;
    strgeninvpreimgs := EGsub`strgeninvpreimgs;
310 bo := EGsub`GR`bo;
    sv := EGsub`GR`sv;
    l := #B;
    u := [];
    for i in [1 .. l] do
315 // strip-like process
        // h fixes base points B[1], B[2], ..., B[i-1]
        pt := PCBFQuotAction(EG, B[i], h);
        if pt notin bo[i] then
            return false, <>;
320 end if;
        u[i] := PCBFQuotSVWordInv(EG, strgenpreimgs,
            strgeninvpreimgs, sv[i], bo[i], pt);
        h := PCBFMult(EG, h, PCBFInv(EG, PCBFWordMultiply(EG,
            strgenpreimgs, strgeninvpreimgs, u[i])));
    end for;
    // if ele is an element of EGsub then its first component is u
325 h := ele;
    word := [];
    for i in [1 .. l] do
        word cat:= WordInverse(u[i]);
    end for;
330 // premultiply h by the inverse of the element encoded by u to
        find the component belonging to EG`N
    h := PCBFMult(EG, PCBFWordMultiply(EG, strgenpreimgs,
        strgeninvpreimgs, word), h);
    if PCBFrho(EG, h) ne Id(EG`GR`G) then return false, <>; end if
    ;
    n := PCBFphi(EG, h);
    if n notin EGsub`N then return false, <>; end if;
335 return true, <u, EGsub`N ! n>;
end function;

PCBFiota := function(EG, ele)
340
    /*

    Arguments:

345 EG: PCBF-group, Type(s): Rec

```

```

    ele: Ordered pair representing an element of EG, Type(s):
    Tup

350
    Parameters:

    Return Type(s):
355
    Tup

    Description:
360
    Returns the image of ele in the super group to which it
    belongs. Do NOT use this method directly. Use PCBFCoerce
    instead.

365
    */

    if EG`ismaster then return ele; end if;
    EGsupergrp := PCBFSuperGrp(EG);
    word := [];
370 for i in [#ele[1] .. 1 by -1] do
        word cat:= ele[1][i];
    end for;
    return PCBFMult(EGsupergrp, PCBFWordMultiply(EGsupergrp, EG`
        strgenpreimgs, EG`strgeninvpreimgs, word), <PCBFId(
        EGsupergrp)[1], EGsupergrp`N ! ele[2]>);
end function;
375

PCBFiotaSeq := function(EG, elts)

    /*
380
    Arguments:

    EG: PCBF-group, Type(s): Rec

385
    elts: Sequence of ordered pairs representing elements of
    EG, Type(s): Seqenum

    Parameters:
390

    Return Type(s):

```

```

395      SeqEnum

      Description:

      Returns a sequence of the respective images (in the super
400 group to which they belong) of each element of elts.

      */

      return [PCBFiota(EG, elts[i]) : i in [1 .. #elts]];
405 end function;

PCBFCoerce := function(EG1, ele, EG2)

410  /*

      Arguments:

      EG1: PCBF-group, Type(s): Rec

415      ele: Ordered pair representing an element of EG1, Type(s):
      Tup

      EG2: PCBF-group, Type(s): Rec

420

      Parameters:

425      Return Type(s):

      BoolElt, Tup

430

      Description:

      Coerces the element, ele, of EG1 into EG2.

      */

435      return PCBFIotaInv(EG2, PCBFiota(EG1, ele));
end function;

440 PCBFin := function(EG1, ele, EG2)

```



```

/*

Arguments :

445   EG1: PCBF-group , Type(s): Rec

      ele: Ordered pair representing an element of EG1, Type(s):
      Tup

450   EG2: PCBF-group , Type(s): Rec

Parameters :

455

Return Type(s):

      BoolElt

460

Description :

      Returns true if and only if the element, ele , of EG1 is an
465   element of EG2.

*/

ans , _ := PCBFCoerce(EG1, ele , EG2);
470   return ans;
end function;

PCBFnotin := function(EG1, ele , EG2)
475
/*

Arguments :

480   EG1: PCBF-group , Type(s): Rec

      ele: Ordered pair representing an element of EG1, Type(s):
      Tup

485   EG2: PCBF-group , Type(s): Rec

Parameters :
```

```

490      Return Type(s):

          BoolElt

495      Description:

          Returns true if and only if the element, ele, of EG1 is not
          an element of EG2.

500      */

      return not PCBFIn(EG1, ele, EG2);
end function;

505

PCBFQuotSubrhoinv := function(EG, H)

    /*
510    Arguments:

          EG: PCBF group, Type(s): Rec

515    H: Subgroup of EG`Q, Type(s): GrpPerm

          Parameters:

520    Return Type(s):

          Rec

525    Description:

          Returns the preimage of the subgroup, H, of EG`Q in EG.

530    */

      Hgens := Setseq(Generators(H));
      gens := PCBFrhoinvSeq(EG, [EG`Q ! Hgens[i] : i in [1 .. #Hgens
          ]]) cat EG`pcgensnf;
      return PCBFsub(EG, gens);
535 end function;

```

```

PCBFSolubleRadical := function(EG)

540  /*
      Arguments:
          EG: PCBF group, Type(s): Rec
545
      Parameters:

550  Return Type(s):
          Rec

555  Description:
          Returns the soluble radical of EG.

      */
560  return PCBFQuotSubrhoinv(EG, SolubleRadical(EG`Q));
end function;

565 PCBFSolvableRadical := function(EG)

      /*
      Arguments:
570  EG: PCBF group, Type(s): Rec

      Parameters:
575

      Return Type(s):
          Rec
580

      Description:

```

```

    Returns the soluble radical of EG.
585    */

    return PCBFSolubleRadical(EG);
end function;
590

PCBFnc1 := function(EG, gens)

    /*
595    Arguments:

        EG: PCBF-group, Type(s): Rec

600    gens: Sequence of ordered pairs representing elements of
        EG, Type(s): SeqEnum or SetIndx

    Parameters:

605

    Return Type(s):

        Rec

610    Description:

        Returns a PCBF record representing the normal closure of
        the subgroup of EG generated by the elements in gens.

615    */

    closurefound := false;
    G := EG;
620    X := PCBFGenerators(G);
    Y := gens;
    H := PCBFsub(G, Y);
    while not closurefound do
        for i in [1 .. BATCHSIZE] do
625            g := PCBFRandom(G);
            _, h := PCBFCoerce(H, PCBFRandom(H), G);
            gen := PCBFConjugate(G, h, g);
            if PCBFnotin(G, gen, H) then
                Append(~Y, gen);
630            H := PCBFsub(G, Y);
            end if;

```

```

        end for ;
        closurefound := true;
        for x in X do
635         for y in Y do
            if PCBFnotin(G, PCBFConjugate(G, y, x), H) then
                closurefound := false;
                break x;
            end if;
640         end for;
        end for;
        end while;
        return H;
    end function;
645

PCBFSubgroupNormalClosure := function(G, H)

    /*
650    Arguments:

        G: PCBF-group, Type(s): Rec
655    H: Subgroup of G, Type(s): Rec

    Parameters:

660    Return Type(s):

        Rec

665    Description:

        Returns a PCBF record representing the normal closure of
        the subgroup H in G.
670    */

    h := PCBFGenerators(H);
    elts := [];
675    for i in [1 .. #h] do
        _, elts[i] := PCBFCoerce(H, h[i], G);
    end for;
    return PCBFncl(G, elts);
end function;

```

680

```
PCBFSubgroupConjugate := function(G, H, g)
```

```
    /*
```

685

```
    Arguments:
```

```
        G: PCBF-group, Type(s): Rec
```

690

```
        H: Subgroup of G, Type(s): Rec
```

```
        g: Element of G, Type(s): Tup
```

695

```
    Parameters:
```

```
    Return Type(s):
```

700

```
    Rec
```

```
    Description:
```

705

```
    Returns a PCBF record representing the conjugate of the  

    subgroup H in G.
```

```
    */
```

710

```
    h := PCBFGenerators(H);
```

```
    elts := [];
```

```
    for i in [1 .. #h] do
```

```
        _, elts[i] := PCBFCoerce(H, h[i], G);
```

```
    end for;
```

715

```
    return PCBFsub(G, [PCBFConjugate(G, elts[i], g) : i in [1 .. #
```

```
        elts]]);
```

```
end function;
```

---

Listing B.10: pcbfderived.m

---

```
/*
```

```
File:
```

```
pcbderived.m
```

4

```
Last modified:
```

```
Sun, 06 Feb 2011 22:14:13 +0000
```

```

9  Author(s):                Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

                                Company:                University of Warwick <http://www.warwick.ac.uk>

14  Description:

                                The methods in this file are used to construct derived subgroups
                                and series
                                of polycyclic-by-finite groups represented by records of type
                                PCBF.

19

                                Notes:

                                MAGMA V2.16-13.

24

                                Copyright 2006-2010, University of Warwick. All rights reserved.

                                */

29

                                load "pcbfsubgroup.m";

34 PCBFCommutatorSubgroup := function(H, K)

                                /*

                                Arguments:

39                                H: PCBF-group, Type(s): Rec

                                K: PCBF-group, Type(s): Rec

44                                Parameters:

                                Return Type(s):

49                                Rec

                                Description:

```

```

54      Returns a PCBF record representing the commutator subgroup
      of H and K. The PCBF-groups H and K are assumed to be
      subgroups of some common PCBF-group. Do NOT use this
      function to compute derived subgroups. The function
59      PCBFDerivedGroup is more efficient.

      */

      EG := PCBFSuperGrp(H);
      X := PCBFGenerators(H);
64      Y := PCBFGenerators(K);
      L := PCBFsub(EG, PCBFiotaSeq(H, X) cat PCBFiotaSeq(K, Y));
      // [H, K] is the normal closure in L of the subgroup generated
      // by [x, y] with x in X, y in Y
      C := [car<PowerSequence(PowerSequence(IntegerRing()))], L`N>
      |];
69      for i in [1 .. #X] do
        -, x := PCBFCoerce(H, X[i], L);
        for j in [1 .. #Y] do
          -, y := PCBFCoerce(K, Y[j], L);
          Append(~C, PCBFCommutator(L, x, y));
74      end for;
      end for;
      return PCBFncl(L, C);
      end function;

79      PCBFDerivedGroup := function(G)

      /*

84      Arguments:

          G: PCBF-group, Type(s): Rec

89      Parameters:

          Return Type(s):

94      Rec

      Description:

99      Returns a PCBF record representing the derived subgroup of

```



```

      G.

      */

104  X := PCBFGenerators(G);
      // G' is the normal closure in G of the subgroup generated by
      (x1, x2) with x1, x2 in X
      return PCBFncl(G, [PCBFCommutator(G, x1, x2) : x1 in X, x2 in
      X]);
  end function;

109  PCBFDerivedSubgroup := function(G)

      /*

114  Arguments:

      G: PCBF-group, Type(s): Rec

      Parameters:

119

      Return Type(s):

      Rec

124  Description:

      Returns a PCBF record representing the derived subgroup of
      G.

129

      */

      return PCBFDerivedGroup(G);
  end function;

134

  PCBFDerivedSeries := function(G)

      /*

139

      Arguments:

      G: PCBF-group, Type(s): Rec

144

      Parameters:

```

```

    Return Type(s):
149      SeqEnum

    Description:
154      Returns a sequence containing the derived series of G.

    */

159  DS := [G];
    H := PCBFDerivedGroup(G);
    i := 1;
    while Order(PCBFQuotGrp(H)) lt Order(PCBFQuotGrp(DS[i])) or
        PCBFNormalSubgrp(H) ne PCBFNormalSubgrp(DS[i]) do
        Append(~DS, H);
164  H := PCBFDerivedGroup(H);
        i += 1;
    end while;
    return DS;
end function;

169

PCBFLowerCentralSeries := function(G)

    /*
174
    Arguments:

        G: PCBF-group, Type(s): Rec

179
    Parameters:

    Return Type(s):
184      SeqEnum

    Description:
189      Returns a sequence containing the lower central series of
        G.

```

```

    */
194   LCS := [G];
      H := PCBFDerivedGroup(G);
      i := 1;
      while Order(PCBFQuotGrp(H)) lt Order(PCBFQuotGrp(LCS[i])) or
        PCBFNormalSubgrp(H) ne PCBFNormalSubgrp(LCS[i]) do
199   Append(~LCS, H);
      H := PCBFCommutatorSubgroup(G, H);
      i += 1;
      end while;
      return LCS;
204 end function;

```

---

Listing B.11: pcbfsylow.m

---

```

1  /*

      File:                pcbfsylow.m

6  Last modified:         Sun, 06 Feb 2011 22:14:13 +0000

      Author(s):           Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

11     Company:            University of Warwick <http://www.warwick.ac.
                           uk>

      Description:

16     The methods in this file are used to construct Sylow subgroups
        of (finite)
        polycyclic-by-finite groups represented by records of type PCBF.

21     Notes:

        MAGMA V2.16-13.

26     Copyright 2006-2010, University of Warwick. All rights reserved.

    */

31 load "pcbderived.m";

```

```

PCBFSylow := function(G, p)

36  /*

    Arguments:

        G: PCBF-group, Type(s): Rec
41
        p: Prime integer, Type(s): RngIntElt

    Parameters:

46

    Return Type(s):

        Rec
51

    Description:

        Returns a Sylow p-subgroup of G.

56
    */

H := PCBFQuotSubrhoinv(G, Sylow(G`Q, p)); // the Sylow p-
      subgroups of G are exactly the Sylow p-subgroups of H
K, pcisodata := PCBFPolycyclicGroup(H);
61 sylowgensK := Setseq(Generators(Sylow(K, p)));
sylowgensG := [car<PowerSequence(PowerSequence(IntegerRing()))
               , G`N> |];
for i in [1 .. #sylowgensK] do
    // map back into K, then into H then into G
    -, sylowgensG[i] := PCBFCoerce(H, PCBFPolycyclicisoinv(H,
        pcisodata, K ! sylowgensK[i]), G);
66 end for;
return PCBFsub(G, sylowgensG);
end function;

71 PCBFSylowSubgroup := function(G, p)

    /*

    Arguments:

76

```

---

```

      G: PCBF-group, Type(s): Rec

      p: Prime integer, Type(s): RngIntElt

81  Parameters:

      Return Type(s):
86  Rec

      Description:
91  Returns a Sylow p-subgroup of G.

      */

96  return PCBFSylow(G, p);
    end function;

PCBFSylowConjugatingElement := function(G, P1, P2)
101 /*

      Arguments:

106  G: PCBF-group, Type(s): Rec

      P1: Sylow p-subgroup of G, Type(s): Rec

      P2: Sylow p-subgroup of G, Type(s): Rec
111

      Parameters:

116  Return Type(s):

      Tup

121  Description:

      Returns an element of G conjugating P1 into P2.

```

```

    */
126 // incomplete
    Q := G`GR`G;
    rhoP1 := PCBFQuotGrp(P1);
    rhoP2 := PCBFQuotGrp(P2);
131 P2N := PCBFQuotSubrhoinv(G, rhoP2); // P2N is finite and
        soluble
    -, z := IsConjugate(Q, rhoP1, rhoP2);
    g := PCBFrhoinv(G, z);
    P1 := PCBFSubgroupConjugate(G, P1, g); // P1 is now a Sylow p-
        subgroup of P2N
    // do the work using power-conjugate presentations
136 L, isodata := PCBFPolycyclicGroup(P2N);
    gens := [];
    for x in PCBFGenerators(P1) do
        -, ele := PCBFCoerce(P1, x, P2N);
        Append(~gens, PCBFPolycycliciso(P2N, L, isodata, ele));
141 end for;
    LP1 := sub<L | gens>; // image of P1 in L
    gens := [];
    for x in PCBFGenerators(P2) do
        -, ele := PCBFCoerce(P2, x, P2N);
146 Append(~gens, PCBFPolycycliciso(P2N, L, isodata, ele));
    end for;
    LP2 := sub<L | gens>; // image of P2 in L
    -, z := IsConjugate(L, LP1, LP2);
    -, y := PCBFCoerce(P2N, PCBFPolycyclicisoinv(P2N, isodata, z),
        G);
151 return PCBFMult(G, g, y);
end function;

```

---

Listing B.12: pcbfcentre.m

---

```

/*
3 File:                pcbfcentre.m

Last modified:         Sun, 06 Feb 2011 22:14:13 +0000

8 Author(s):           Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

Company:              University of Warwick <http://www.warwick.ac.
                        uk>

```

*Description :*

*The methods in this file are used to compute the centre of a*  
 18 *polycyclic-by-finite group represented by records of type PCBF.*

*Notes :*

23 *MAGMA V2.16–13.*

*Copyright 2006–2010, University of Warwick. All rights reserved.*

28 *\*/*

**load** "pcbfsylow.m";

33 **PCBFCentre** := **function**(G)

*/\**

38 *Arguments :*

*G: PCBF-group, Type(s): Rec*

43 *Parameters :*

*Return Type(s):*

48 *Rec*

*Description :*

53 *Returns the centre of G.*

*\*/*

**S** := **PCBFSolubleRadical**(G);  
 58 **K**, **pcisodata** := **PCBFPolycyclicGroup**(S); *// pcisodata encodes*  
*an isomorphism from S onto P*  
**C** := **Centre**(K);  
**if** **IsTrivial**(C) **then return** **PCBFsub**(G, []); **end if**;

```

centregensG := [car<PowerSequence(PowerSequence(IntegerRing()))
    ), G`N> |];
if PCBFIsSoluble(G) then
63 // C is the centre of G
centregensC := Setseq(Generators(C));
for i in [1 .. #centregensC] do
    // map back into K, then into S and then into G
    -, centregensG[i] := PCBFCoerce(S, PCBFPolycyclicisoinv(S,
        pcisodata, K ! centregensC[i]), G);
68 end for;
return PCBFsub(G, centregensG);
end if;
AC, kappa := AbelianGroup(C); // kappa: C -> AC
invars := Invariants(AC);
73 r := #invars;
Z := Integers();
mats := [MatrixAlgebra(Z, r) | ];
I := IdentityMatrix(Z, r);
O := ZeroMatrix(Z, r, r);
78 // for each generator of G, calculate the matrix specifying
    conjugation action of that generator on AC, and subtract
    the identity matrix
subjgens := PCBFGenerators(G);
matseq := [];
for i in [1 .. #subjgens] do
    for j in [1 .. r] do
83 -, g := PCBFCoerce(S, PCBFPolycyclicisoinv(S, pcisodata, K
        ! (AC.j @@ kappa)), G);
        -, s := PCBFCoerce(G, PCBFConjugate(G, g, subjgens[i]), S)
        ;
        matseq[j] := Eltseq((C ! PCBFPolycycliciso(S, K, pcisodata
            , s)) @ kappa);
    end for;
    mat := Matrix(matseq) - I;
88 if mat ne O then Append(~mats, mat); end if;
end for;
if #mats eq 0 then
    // C is the centre of G
    centregensC := Setseq(Generators(C));
93 for i in [1 .. #centregensC] do
        // map back into K, then into S then into G
        -, centregensG[i] := PCBFCoerce(S, PCBFPolycyclicisoinv(S,
            pcisodata, K ! centregensC[i]), G);
    end for;
    return PCBFsub(G, centregensG);
98 end if;
// the elements in the centre of G are basically those in the
    nullspace of each of the matrices. But the matrices are

```



```

    over the integers, and solutions modulo  $\text{invars}[i]$  are
    required. To handle this, extra rows are adjoined to the
    matrices
    invarmat := O;
    for i in [1 .. r] do invarmat[i][i] := invars[i]; end for;
    modmats := [];
103   for i in [1 .. #mats] do
        modmats[i] := VerticalJoin([mats[i]] cat [O : j in [1 ..
            i - 1]] cat [invarmat] cat [O : j in [i + 1 .. #mats
                ]]);
    end for;
    // now the centre corresponds to the intersection of the
    nullspaces of the matrices – equivalently, the nullspace of
    their horizontal join
    nsmat := NullspaceMatrix(HorizontalJoin(modmats));
108   centregensC := [C | ];
    for i in [1 .. Nrows(nsmat)] do
        a := AC ! Eltseq(ExtractBlock(nsmat, i, 1, 1, r));
        if a ne AC ! 0 then
            Append(~centregensC, a @@ kappa);
113   end if;
    end for;
    for i in [1 .. #centregensC] do
        // map back into K, then into S and then into G
        -, centregensG[i] := PCBFCoerce(S, PCBFPolycyclicisoinv(S,
            pcisodata, K ! centregensC[i]), G);
118   end for;
    return PCBFsub(G, centregensG);
end function;

123 PCBFCenter := function(G)

    /*

    Arguments:

128   G: PCBF-group, Type(s): Rec

    Parameters:

133

    Return Type(s):

    Rec

138

```

*Description :*

*Returns the centre of G.*

143

*\*/*

**return** PCBFCentre(G);  
**end function**;

---

Listing B.13: pcbfcentraliser.m

---

*/\**

2

*File :* *pcbfccentraliser-test.m*

*Last modified :* *Sun, 06 Feb 2011 22:14:13 +0000*

7

*Author(s) :* *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

12 *Company :* *University of Warwick <http://www.warwick.ac.uk>*

*Description :*

17 *The methods in this file are used to compute the centraliser of*  
*an element*  
*belonging to a polycyclic-by-finite group represented by records*  
*of type*  
*PCBF.*

22 *Notes :*

*MAGMA V2.16-13.*

27 *Copyright 2006-2010, University of Warwick. All rights reserved.*

*\*/*

32 **load** "pcbfccentre.m";

PCBFCentraliser := **function**(G, g)

```

37  /*

    Arguments:

        G: PCBF-group, Type(s): Rec
42    g: Element of G, Type(s): Tup

    Parameters:

47

    Return Type(s):

        Rec
52

    Description:

        Returns the centraliser of g in G.

57  */

    if PCBFeq(G, g, PCBFId(G)) then return G; end if; // cheap
        check for g = PCBFId(G)
    Q := G`GR`G;
62  c := PCBFrhoInvSeq(G, Setseq(Generators(Centraliser(Q, PCBFrho
        (G, g)))))) cat G`pcgensnf; // generators of inverse image
        of centraliser of g in the quotient
    C := PCBFsub(G, c);
    N := ElementaryAbelianSeriesCanonical(G`N); // the canonical
        version of the elementary abelian series function is used
        as a sequence of normal subgroups is required
    t := #N;
    // the centraliser of g in G/N[1] has been computed, so it's
        time to move downward through the elementary abelian series
67  for i in [1 .. t - 1] do
        M, phi := GModule(N[1], N[i], N[i + 1]); // phi : N[i] -> M
        F := Field(M);
        d := Dimension(M);
        A := GL(d + 1, F);
72  I := Id(A);
        bas := [PCBFphiInv(G, (M.k)@@phi) : k in [1 .. d]];
        cimgs := [A | ];
        for j in [1 .. #c] do
            y := [];
77  for k in [1 .. d] do

```

```

      // computing k-th row of the image of c[i] under the
      affine map
      y[k] := Eltseq((PCBFphi(G, PCBFConjugate(G, bas[k], c[j]
      )))@phi) cat [Zero(F)];
    end for;
    y[d + 1] := Eltseq((PCBFphi(G, PCBFCommutator(G, g, c[j]))
    )@phi) cat [Id(F)];
82    cimgs[j] := (A ! Matrix(y)); // these images should really
      be used to define a homomorphism, but homomorphisms
      involving PCBF groups have not yet been implemented
      internally in MAGMA
  end for;
  defcimgs := [A | ];
  cnew := [car<PowerSequence(PowerSequence(IntegerRing())) , G`
  N> |];
  indexmap := [];
87  // clean generating set
  for j in [1 .. #cimgs] do
    if cimgs[j] eq I then
      Append(~cnew, c[j]);
      continue;
92    end if;
    for k in [1 .. #defcimgs] do
      if cimgs[j] eq defcimgs[k] then
        Append(~cnew, PCBFMult(G, c[indexmap[k]], PCBFInverse(
        G, c[j])));
        continue j;
97    end if;
    end for;
    // cimgs[j] is a genuine new generator
    Append(~defcimgs, cimgs[j]);
    Append(~indexmap, j); // this sequence is needed to
      correspond the position of cimgs[j] in defcimgs to j
102  end for;
  if defcimgs eq [] then continue; end if;
  B := sub<A | defcimgs>; // range of alpha
  // compute stabliser in B of (0, ..., 0, 1)
  V := VectorSpace(A);
107  trickvec := V.(d + 1);
  S := Stabiliser(B, trickvec);
  s := Setseq(Generators(S));
  // map S back into G
  // the code that follows is slow and should be rewritten
  when homomorphisms involving PCBF-groups become available
112  L, psi := FPGGroup(B);
  // to map S back into G, the preimage kernel of psi in G is
  needed
  rels := Relations(L);

```

```

kerpsipreimggens := [];
for j in [1 .. #rels] do
117   ele := PCBFId(G);
   word := Eltseq(LHS(rels[j])*(RHS(rels[j])^-1));
   // multiply word in G
   for k in [1 .. #word] do
       ele := PCBFMult(G, ele, word[k] lt 0 select PCBFInverse(
           G, c[indexmap[-word[k]]]) else c[indexmap[word[k]]]);
122   end for;
   _, kerpsipreimggens[j] := PCBFCoerce(G, ele, C);
end for;
kerpsipreimg := PCBFnc(C, kerpsipreimggens);
kerpsipreimggens := PCBFGenerators(kerpsipreimg);
127 offset := #cnew;
for j in [1 .. #kerpsipreimggens] do
   _, cnew[offset + j] := PCBFCoerce(C, kerpsipreimggens[j],
       G);
end for;
for j in [1 .. #s] do
132   ele := PCBFId(G);
   word := Eltseq(s[j]@@psi);
   // multiply word in G
   for k in [1 .. #word] do
       ele := PCBFMult(G, ele, word[k] lt 0 select PCBFInverse(
           G, c[indexmap[-word[k]]]) else c[indexmap[word[k]]]);
137   end for;
   Append(~cnew, ele);
end for;
c := cnew;
C := PCBFsub(G, c);
142 end for;
return C;
end function;

```

```

147 PCBFCentralizer := function(G, g)

```

/\*

*Arguments:*

152

*G: PCBF-group, Type(s): Rec*

*g: Element of G, Type(s): Tup*

157

*Parameters:*

```

162      Return Type(s):
          Rec

      Description:
167      Returns the centraliser of g in G.

      */
172  return PCBFCentraliser(G, g);
end function;

```

---



---

Listing B.14: pcbfconjugacy.m

---

```

/*
2  File:                pcbfcconjugacy.m

      Last modified:    Sun, 06 Feb 2011 22:14:13 +0000
7
      Author(s):        Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>

12 Company:            University of Warwick <http://www.warwick.ac.uk>

      Description:

17 The methods in this file are used to test conjugacy of elements
    belonging
    to a polycyclic-by-finite group represented by records of type
    PCBF.

      Notes:
22 MAGMA V2.16–13.

      Copyright 2006–2010, University of Warwick. All rights reserved.
27 */

```

```

load "pcbfcentraliser.m";
32

PCBFIsConjugate := function(G, g, h)

    /*
37    Arguments:

        G: PCBF-group, Type(s): Rec

42    g: Element of G, Type(s): Tup

        h: Element of G, Type(s): Tup

47    Parameters:

        Return Type(s):

52    BoolElt, Rec

        Description:

57    Returns true, and an element conjugating g into h, if g and
        h are conjugate in G, false otherwise.

    */

62    if PCBFeq(G, g, h) then return true, PCBFId(G); end if; //
        cheap check for g = h
    Q := G`GR`G;
    N := ElementaryAbelianSeriesCanonical(G`N); // the canonical
        version of the elementary abelian series function is used
        as a sequence of normal subgroups is required
    t := #N;
    gbar := PCBFrho(G, g);
67    hbar := PCBFrho(G, h);
    isconj, zbar := IsConjugate(Q, gbar, hbar);
    if not isconj then return false, -; end if; // g and h aren't
        even conjugate in the quotient Q
    z := PCBFrhoInv(G, zbar); // preimage of conjugating element
    h := PCBFConjugate(G, h, PCBFInv(G, z)); // h is now equal to
        g in the quotient G/N

```

```

72  c := PCBFrhoInvSeq(G, Setseq(Generators(Centraliser(Q, PCBFrho
      (G, g)))))) cat G'pcgensnf; // generators of inverse image
      of centraliser of g in the quotient
C := PCBFsub(G, c);
// the centraliser of g in G/N[1] has been computed, so it's
      time to move downward through the elementary abelian series
for i in [1 .. t - 1] do
    ele := PCBFId(G);
77  M, phi := GModule(N[1], N[i], N[i + 1]); // phi : N[i] → M
    F := Field(M);
    d := Dimension(M);
    A := GL(d + 1, F);
    I := Id(A);
82  bas := [PCBFphiInv(G, (M.k)@@phi) : k in [1 .. d]];
    cimgs := [A | ];
    for j in [1 .. #c] do
        y := [ ];
        for k in [1 .. d] do
87          // computing k-th row of the image of c[i] under the
              affine map
          y[k] := Eltseq((PCBFphi(G, PCBFConjugate(G, bas[k], c[j]
              )))@phi) cat [Zero(F)];
        end for;
        y[d + 1] := Eltseq((PCBFphi(G, PCBFCommutator(G, g, c[j]))
            )@phi) cat [Id(F)];
        cimgs[j] := (A ! Matrix(y)); // these images should really
            be used to define a homomorphism, but homomorphisms
            involving PCBF groups have not yet been implemented
            internally in MAGMA
92  end for;
    defcimgs := [A | ];
    cnew := [car<PowerSequence(PowerSequence(IntegerRing()))], G`
        N> |];
    indexmap := [ ];
    // clean generating set
97  for j in [1 .. #cimgs] do
        if cimgs[j] eq I then
            Append(~cnew, c[j]);
            continue;
        end if;
102  for k in [1 .. #defcimgs] do
            if cimgs[j] eq defcimgs[k] then
                Append(~cnew, PCBFMult(G, c[indexmap[k]], PCBFInverse(
                    G, c[j])));
                continue j;
            end if;
107  end for;
    // cimgs[j] is a genuine new generator

```



```

Append(~defcimgs, cimgs[j]);
Append(~indexmap, j); // this sequence is needed to
correspond the position of cimgs[j] in defcimgs to j
end for;
112 if defcimgs eq [] then
    // if defcimgs is empty, then the new centraliser is equal
    to the old centraliser
    // if g and h are conjugate, then they should already be
    equal in  $N[i + 1]$ 
    if PCBFphi(G, PCBFMult(G, PCBFInv(G, g), h)) notin N[i +
        1] then
        return false, -;
117 end if;
    continue;
end if;
B := sub<A | defcimgs>; // range of alpha
L, psi := FPGGroup(B); // image of B as a finitely presented
group
122 // compute stabliser in B of (0, ..., 0, 1)
V := VectorSpace(A);
orbvec := V.(d + 1);
S := Stabiliser(B, orbvec);
O := Orbit(B, orbvec);
127 tau, P := OrbitAction(B, O); // P is a permutation group
which gives the action of B on O, tau : B --> P
testvec := V ! (Eltseq((PCBFphi(G, PCBFMult(G, PCBFInv(G, g),
    h)))@phi) cat [Id(F)]); // this vector will be tested
for membership in the orbit O
if testvec notin O then
    return false, -;
end if;
132 isconj, zperm := IsConjugate(P, tau(orbvec), tau(testvec));
if not isconj then
    return false, -;
end if;
ele := PCBFId(G);
137 word := Eltseq((zperm@@tau)@@psi);
// find preimage in C of (local) conjugating element
for k in [1 .. #word] do
    ele := PCBFMult(G, ele, word[k] lt 0 select PCBFInverse(G,
        c[indexmap[-word[k]]]) else c[indexmap[word[k]]]);
end for;
142 h := PCBFConjugate(G, h, PCBFInv(G, ele)); // h is now equal
to g in the quotient  $G/N[i + 1]$ 
z := PCBFMult(G, ele, z); // modify (global) conjugating
element
// map S back into G

```

```

// the code that follows is slow and should be rewritten
// when homomorphisms involving PCBF-groups are implemented
// to map S back into G, the preimage of the kernel of psi
// in G is needed
147 rels := Relations(L);
kerpsipreimggens := [];
for j in [1 .. #rels] do
    ele := PCBFId(G);
    word := Eltseq(LHS(rels[j]) * (RHS(rels[j])^-1));
152 // multiply word in G
    for k in [1 .. #word] do
        ele := PCBFMult(G, ele, word[k] lt 0 select PCBFInverse(
            G, c[indexmap[-word[k]]]) else c[indexmap[word[k]]]);
    end for;
    _, kerpsipreimggens[j] := PCBFCoerce(G, ele, C);
157 end for;
kerpsipreimg := PCBFnc(C, kerpsipreimggens);
kerpsipreimggens := PCBFGenerators(kerpsipreimg);
offset := #cnew;
for j in [1 .. #kerpsipreimggens] do
162 _, cnew[offset + j] := PCBFCoerce(C, kerpsipreimggens[j],
    G);
end for;
s := Setseq(Generators(S));
for j in [1 .. #s] do
    ele := PCBFId(G);
167 word := Eltseq(s[j]@@psi);
    // multiply word in G
    for k in [1 .. #word] do
        ele := PCBFMult(G, ele, word[k] lt 0 select PCBFInverse(
            G, c[indexmap[-word[k]]]) else c[indexmap[word[k]]]);
    end for;
172 Append(~cnew, ele);
end for;
c := cnew;
C := PCBFsub(G, c);
end for;
177 return true, z;
end function;

```

Listing B.15: pcbfmain.m

```

/*
2
File:                pcbfmain.m

Last modified:       Sun, 06 Feb 2011 22:14:13 +0000

```

*Author(s):*                    *Shavak Sinanan <S.K.Sinanan@warwick.ac.uk>*

<sup>12</sup> *Company:*                    *University of Warwick <http://www.warwick.ac.uk>*

*Description:*

<sup>17</sup> *Root file for the PCBF package.*

*Notes:*

<sup>22</sup> *MAGMA V2.16–13.*

*Copyright 2006–2010, University of Warwick. All rights reserved.*

<sup>27</sup> *\*/*

**load** "pcbfconjugacy.m";  
**load** "pcbfexamples.m";

<sup>32</sup>

PCBFNormalForm := **function**(EG, e)

*/\**

<sup>37</sup>

*Arguments:*

*EG: PCBF-group, Type(s): Rec*

<sup>42</sup> *e: element of original polycyclic-by-finite group, Type(s): GrpElt*

*Parameters:*

<sup>47</sup>

*Return Type(s):*

*GrpPermElt*

<sup>52</sup>

*Description:*

---

```

    Computes the normal form of the element y in the PCBF-group
57    EG. USED FOR TESTING ONLY.

    */

    g := EG`rho(e);
62    word := StrongGenWord(EG`GR, g);
    return <StrongGenNormalForm(GR, g), EG`phi(WordMultiply(EG`
        strgenpreimgs, EG`strgeninvpreimgs, word)^-1 * e)>;
end function;

67 function PCBFRevert(EG, ele)

    /*

    Arguments:

72    EG: PCBF-group, Type(s): Rec

        ele: ordered pair representing an element of the PCBF-group
        EG, Type(s): Tup
77

    Parameters:

82    Return Type(s):

        GrpElt

87    Description:

        Returns the element of the original master group
        represented by ele. USED FOR TESTING ONLY.

92    */

    if not EG`ismaster then
        // recurse if the given group is not a master group
        return PCBFRevert(EG`supergrp, PCBFiota(EG, ele));
97    end if;
    phi := EG`phi;
    word := [];
    for i in [#ele[1] .. 1 by -1] do
        word cat:= ele[1][i];

```

```

102  end for ;
      ans := WordMultiply(EG`strgenpreims , EG`strgeninvpreims ,
        word) ;
      ans *:= ele [2] @@phi;
      return ans;
end function;
107

function PCBFRevertSeq(EG, elts)

  /*
112  Arguments:

      EG: PCBF-group , Type(s): Rec

117  elts: sequence of ordered pairs representing elements of
      the PCBF-group EG, Type(s): Tup

      Parameters:

122  Return Type(s):

      SeqEnum
127

      Description:

      Returns a sequence of elements of the original master group
132 corresponding to elts. USED FOR TESTING ONLY.

      */

      return [PCBFRevert(EG, elts[i]) : i in [1 .. #elts]];
137 end function;

function PCBFRevertSub(EG)

142  /*

      Arguments:

      EG: PCBF-group , Type(s): Rec
147

```

*Parameters:*

152 *Return Type(s):*

*Grp*

157 *Description:*

*Returns a subgroup of  $EG^E$  isomorphic to  $EG$ .  
USED FOR TESTING ONLY.*

162 *\*/*

```
return sub< $EG^E$  | PCBFRevertSeq(EG, PCBFGenerators(EG))>;  
end function;
```

---

# Bibliography

- Baumslag, Gilbert, Frank B. Cannonito, Derek J. S. Robsinson and Dan Segal (1991), ‘The Algorithmic Theory of Polycyclic-by-Finite Groups’, *Journal of Algebra* **142**, 118–149.
- Bosma, Wieb, John Cannon and Catherine Playoust (1997), ‘The Magma algebra system. I. The user language.’, *J. Symbolic Comput.* **24(3-4)**, 235–265.
- Chandler, Bruce and Wilhelm Magnus (1982), *The History of Combinatorial Group Theory: A Case Study in the History of Ideas*, Springer.
- Dixon, J. D. (1982), ‘Exact solution of linear equations using  $p$ -adic expansion’, *Numerische Mathematik* **40**, 137–141.
- Eick, Bettina (2001), Algorithms for Polycyclic Groups, Habilitationsschrift, Universität Kassel.
- Haramoto, H. and M. Matsumoto (2009), ‘A  $p$ -adic algorithm for computing the inverse of integer matrices’, *Journal of Computational and Applied Mathematics* **225**, 320–322.
- Holt, Derek F., Bettina Eick and Eamonn A. O’Brien (2005), *Handbook of Computational Group Theory*, Discrete Mathematics and its Applications, Chapman & Hall/CRC.
- James, Gordon and Martin Liebeck (2001), *Representations and Characters of Groups*, Cambridge University Press.

- Kleidman, Peter and Martin Liebeck (1990), *The Subgroup Structure of the Finite Classical Groups*, Vol. 129 of *London Math. Soc. Lecture Note Series*, Cambridge University Press.
- Robinson, Derek J. S. (1996), *A Course in the Theory of Groups*, Second edn, Springer.
- Rotman, Joseph J. (1995), *An Introduction to the Theory of Groups*, Fourth edn, Springer.
- Rotman, Joseph J. (2002), *Advanced Modern Algebra*, Prentice-Hall, Pearson Education.
- Segal, Dan (1983), *Polycyclic Groups*, Cambridge University Press.
- Seress, Ákos (2003), *Permutation Group Algorithms*, Vol. 152 of *Cambridge Tracts in Mathematics*, Cambridge University Press.
- Sims, Charles C. (1970), Computational methods in the study of permutation groups, in J. Leech, ed., 'Computational Problems in Abstract Algebra', Pergamon Press, Oxford, pp. 169–183.
- Sims, Charles C. (1971), Computation with permutation groups, in 'Proc. Second Symposium on Symbolic and Algebraic Manipulation', ACM Press, New York, pp. 23–28.
- Sims, Charles C. (1994), *Computation with finitely presented groups*, Vol. 48 of *Encyclopedia of Mathematics and its Applications*, Cambridge University Press.